AD-A054 357 TECHNOLOGY SERVICE CORP SANTA MONICA CALIF F/G 17/9
MULTIDOMAIN ALGORITHM EVALUATION. VOLUME I.(U)
APR 78 W C LILES, J C DEMMEL, I S REED F30602-76-C-0319
UNCLASSIFIED TSC-PD-B525-1-VOL-1 RADC-TR-78-59-VOL-1 NL

1 OF 3
AD
A054357

FOR FURTHER TRAN

AD A054357

RADC-TR-78-59, Volume I (of two)
Final Technical Report
April 1978

# MULTIDOMAIN ALGORITHM EVALUATION

William C. Liles
James C. Demmel
Irving S. Reed
John D. Mallett
Lawrence E. Brennan

Technology Service Corporation

AD No.
DDC FILE COPY

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

DDC
RECEIVED
MAY 30 1978
D

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-59, Volume I (of two) has been reviewed and is approved for publication.

APPROVED: Vincent Vannicola

VINCENT VANNICOLA
Project Engineer

APPROVED: Joseph L. Ryerson

JOSEPH L. RYERSON
Technical Director
Surveillance Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (OCTS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | |
|---|---|
| 1. REPORT NUMBER: RADC-TR-78-59, Vol I (of two) - Vol-I | 2. GOVT ACCESSION NO. |
| 3. RECIPIENT'S CATALOG NUMBER | |
| 4. TITLE (and Subtitle): MULTIDOMAIN ALGORITHM EVALUATION. Volume I. | 5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. 25 Jun 76 - 18 Oct 77 |
| | 6. PERFORMING ORG. REPORT NUMBER: TSC-PD-B525-1-Vol-I |
| 7. AUTHOR(s): William C. Liles, James C. Demmel, Irving S. Reed, John D. Mallett, Lawrence E. Brennan | 8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0319 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: Technology Service Corporation, 2811 Wilshire Boulevard, Santa Monica CA 90403 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 45060196, 01 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (OCTS), Griffiss AFB NY 13441 | 12. REPORT DATE: April 1978 |
| | 13. NUMBER OF PAGES: 271 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same (12) 274p. | 15. SECURITY CLASS. (of this report): UNCLASSIFIED |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Vincent Vannicola (OCTS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Adaptive array radar
Adaptive algorithms
Associative processors
Parallel processors
Vector pipeline processors
Matrix inversion

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this study is to evaluate different algorithms for solving for up to 200 adaptive weights in an adaptive array radar, using the sample covariance matrix inversion technique. The sample covariance matrix inversion technique was studied because of its ability to handle adaptation in many domains, i.e., spatial, temporal, and polarization. → next Page

(Cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

404 432

Item 20 (Cont'd)

The algorithms and their implementations on different computer architectures, such as associative, parallel, vector pipeline, and sequential, are considered. Both theoretical timings and actual timings on currently available machines are obtained.

Major conclusions reached are that 1) because of the strong dependence of an algorithm's implementation on the computer architecture, it is not possible to choose the best algorithm by operation counts; 2) it is possible to greatly improve system performance by using separate processors to perform the covariance matrix computation and weight calculations; 3) parallel complex arithmetic implemented in hardware would greatly improve a system's performance; and 4) associativity is not useful for this problem.

Finally, an architecture designed specifically for solving for adaptive weights is outlined.

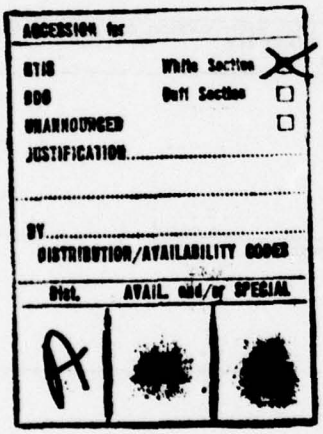


TABLE OF CONTENTS


| Section | | | Page |
|---|---|---|---|
| List of Figures | | | v |
| List of Tables | | | ix |
| 1.0 | INTRODUCTION | | 1 |
| | 1.1 | STATEMENT OF THE PROBLEM | 1 |
| | 1.2 | RESTATEMENT OF THE PROBLEM IN MATHEMATICAL TERMS | 2 |
| | 1.3 | APPROACH | 3 |
| | 1.4 | CONCLUSIONS AND SUMMARY | 5 |
| 2.0 | RADAR CONSIDERATIONS | | 13 |
| | 2.1 | RADAR OPERATING ENVIRONMENT | 13 |
| | 2.2 | THEORY OF ADAPTIVE RADAR | 15 |
| | 2.3 | RADAR ENGINEERING CONSIDERATIONS | 21 |
| | 2.4 | A SAMPLE VOLTAGE VECTOR MODEL | 22 |
| 3.0 | MATHEMATICAL TECHNIQUES | | 25 |
| | 3.1 | STATEMENT OF THE PROBLEM, NOTATION, APPROACH, AND ASSUMPTIONS | 25 |
| | 3.2 | ALGORITHMS FOR SOLVING $MW = \bar{S}$ | 29 |
| | | 3.2.1 Introduction | 29 |
| | | 3.2.2 Complex Arithmetic | 34 |
| | | 3.2.3 Inverse Matrix Update (IMU) Algorithm | 35 |
| | | 3.2.4 Updating the Sample Covariance Matrix | 37 |
| | | 3.2.5 Direct Methods of Solving $MW = \bar{S}$ | 42 |
| | | 3.2.6 Iterative Methods of Solving $MW = \bar{S}$ | 47 |
| | | 3.2.7 Parallel Methods | 52 |
| | | 3.2.8 The Gram-Schmidt Technique | 54 |
| | | 3.2.9 Operation Counts and Conclusions | 56 |
| 4.0 | IMPLEMENTATIONS | | 59 |
| | 4.1 | INTRODUCTION | 59 |
| | 4.2 | SEQUENTIAL PROCESSORS | 61 |

TABLE OF CONTENTS (Cont'd)


| Section | | | Page |
|---|---|---|---|
| | 4.2.1 | Introduction to Sequential Processors . . . . . | 61 |
| | 4.2.2 | Implementations of Algorithms to Determine Weights on a Sequential Processor . . . . . | 62 |
| | 4.2.3 | An Implementation of an Algorithm for Determining Adaptive Weights on a Sequential Processor--the CDC 7600 . . . . . | 69 |
| 4.3 | VECTOR PIPELINE PROCESSORS . . . . . | | 91 |
| | 4.3.1 | Introduction to Vector Pipeline Processors . . . | 91 |
| | 4.3.2 | Implementations of Algorithms to Determine Weights For A Vector Pipeline Processor . . . . . | 104 |
| | 4.3.3 | Implementations of Algorithms for Determining Adaptive Weights on Vector Pipeline Processors . . | 139 |
| | 4.3.4 | Conclusions for Vector Pipeline Processors . . . | 183 |
| 4.4 | IMPLEMENTATIONS TO DETERMINE WEIGHTS FOR A PARALLEL PROCESSOR . . . . . | | 185 |
| | 4.4.1 | Introduction . . . . . | 185 |
| | 4.4.2 | PEPE . . . . . | 188 |
| | 4.4.3 | ILLIAC IV . . . . . | 221 |
| | 4.4.4 | STARAN . . . . . | 226 |
| 4.5 | HYPOTHETICAL SYSTEM . . . . . | | 247 |
| | 4.5.1 | Calculation of M . . . . . | 247 |
| | 4.5.2 | Solving $MW = \overline{S}$ . . . . . | 249 |
| BIBLIOGRAPHY . . . . . | | | 253 |

## LIST OF FIGURES


| Number | | Page |
|---|---|---|
| 1.1a | Results of Radar Simulation with One Jammer for 200-Weight Adaptive Array Using the Sample Covariance Matrix Inverse Method . . . . . | 7 |
| 1.1b | Results of Radar Simulation with Two Jammers for 200-Weight Adaptive Array Using the Sample Covariance Matrix Inverse Method . . . . . | 8 |
| 1.2 | Implementation Timing Comparisons (for 2N Samples) . . . . . | 9 |
| 3.1 | Summary of Sequential Mathematical Techniques . . . . . | 32 |
| 4.1 | Different Algorithms for Computing T = B*C + E + G*H + R On a Sequential Processor with Instruction Overlap . . . . . | 63 |
| 4.2 | Different Algorithms for Computing $S = \sum_{J=1}^{N} A(J)$ on a Sequential Processor with Instruction Overlap . . . . . | 63 |
| 4.3a | Sequential Processor Storage Scheme for the Sample Voltage Vector X: $X_j = X_{Rj} + iX_{Ij}$, $1 \le j \le N$ . . . . . | 67 |
| 4.3b | Sequential Processor Storage Scheme for the Sample Covariance Matrix M: $M_{j,K} = M_{Rj,K} + M_{Ij,K}$, $1 \le j$, $k \le N$ . . . . . | 67 |
| 4.3c | Sequential Processor Storage Scheme for the Steering Vectors $S^{(\ell)}$, $S_j^{(\ell)} = S_{Rj}^{(\ell)} + i\, S_{Ij}^{(\ell)}$, $1 \le j \le N$, $1 \le \ell \le K$ . . . . . | 68 |
| 4.4 | Basic CDC 7600 System Configuration with Semiconductor Memory . . . . . | 70 |
| 4.5 | CDC 7600 Decomposition Times . . . . . | 84 |
| 4.6 | CDC 7600 First Back Substitution Times . . . . . | 85 |
| 4.7 | CDC 7600 Second Back Substitution Times . . . . . | 86 |
| 4.8 | CDC 7600 Sample Covariance Matrix Update Times . . . . . | 87 |
| 4.9 | CDC 7600 Total Times to Process 2N Samples . . . . . | 88 |

## LIST OF FIGURES (Cont'd)


| Number | | Page |
|---|---|---|
| 4.10 | Timing Scheme of a Pipeline Processor . . . . . | 92 |
| 4.11 | Timing Scheme for the Following Sequence of Operations Using Chaining: $T_i = C_i/D_i$, $U_i = B_i * T_i$, $S_i = A_i + U_i$, $i = 1,\ldots,6$ . . . . . | 102 |
| 4.12a | Timing Scheme for the Following Sequence of Operations Using Chaining: $T_i = C_i * D_i$, $U_i = B_i + T_i$, $E_i = U_i * A_i$, $i = 1,\ldots,6$, $E_i = (B_i + (C_i * D_i)) * A_i$ . . . . . | 102 |
| 4.12b | Storage Scheme Needed to Perform Sequence of Operations in Figure 4.12a . . . . . | 102 |
| 4.13 | Summing the Product of Two Complex Vectors $S_R + iS_j = \sum_{j=1}^{n} (A_j + iB_j) * (C_j + iD_j)$ . . . . . | 107 |
| 4.14 | Storage Methods for Hermitian Matrices . . . . . | 110 |
| 4.15a | Multiplying a Packed Matrix by a Vector . . . . . | 112 |
| 4.15b | Multiplying an Unpacked, Filled Matrix by a Vector . . . . . | 112 |
| 4.15c | Multiplying an Unpacked Half-Filled Matrix by a Vector . . . | 112 |
| 4.16 | Storage Schemes of "Stretched" Input Voltage Vector $X = X_R + iX_I$ Required by Implementation 2 for a Sample Covariance Matrix Calculation When N=4 . . . . . | 119 |
| 4.17a | Componentwise Storage of $N_s$ Sample Vectors . . . . . | 120 |
| 4.17b | Vectorwise Storage of $N_s$ Sample Vectors . . . . . | 120 |
| 4.18 | Storage Schemes for the Factor L* of the Matrix M . . . . . | 124 |
| 4.19a | Packed Storage Scheme for Componentwise Augmentation of M when N = 4 and K = 3 . . . . . | 125 |
| 4.19b | Unpacked Storage Scheme for Componentwise Augmentation of M when N = 4 and K = 3 . . . . . | 125 |
| 4.20 | Abbreviations for Different Storage Schemes Used in Table 4.8 and Appendix B . . . . . | 126 |
| 4.21 | Basic CDC STAR-100 Configuration . . . . . | 139 |

LIST OF FIGURES (Cont'd)


| Number | | Page |
|---|---|---|
| 4.22 | Operand Formats | 141 |
| 4.23 | Floating-Point Pipe 1 | 142 |
| 4.24 | Floating-Point Pipe 2 | 142 |
| 4.25 | CDC STAR-100 Decomposition Times | 154 |
| 4.26 | CDC STAR-100 First Back Substitution Times | 155 |
| 4.27 | CDC STAR-100 Second Back Substitution Times | 156 |
| 4.28 | CDC STAR-100 Update Sample Covariance Matrix Times | 157 |
| 4.29 | CDC STAR-100 Total Times to Process 2N Samples | 158 |
| 4.30 | Results of Radar Simulation for 200-Weight Adaptive Array Using the Sample Covariance Matrix Inverse Method | 160 |
| 4.31 | Basic CRAY-1 Organization | 162 |
| 4.32 | CRAY-1 Floating-Point Data Format | 169 |
| 4.33 | CRAY-1 Decomposition Times | 176 |
| 4.34 | CRAY-1 First Back Substitution Times | 177 |
| 4.35 | CRAY-1 Second Back Substitution Times | 178 |
| 4.36 | CRAY-1 Update Sample Covariance Matrix Time | 179 |
| 4.37 | CRAY-Total Times to Process 2N Samples | 180 |
| 4.38 | PEPE Architecture Block Diagram | 189 |
| 4.39a | Storage Scheme for the Sample Covariance Matrix on the PEPE | 196 |
| 4.39b | Storage Schemes of the Factored Sample Covariance Matrix on the PEPE | 196 |
| 4.39c | Storage Schemes for K Steering Vectors on the PEPE | 197 |
| 4.39d | Abbreviations for Different Combinations of Factored Sample Covariance Matrix and Steering Vector Storage Schemes on the PEPE | 197 |

LIST OF FIGURES (Cont'd)


| Number | | Page |
|---|---|---|
| 4.39e | Storage Scheme Required for Augmented Decomposition on the PEPE | 198 |
| 4.40a | Transposing the r x c Matrix M on a PEPE | 201 |
| 4.40b | Transposing a Rectangular Matrix by Method 1 on a PEPE | 201 |
| 4.40c | Transposing a Rectangular Matrix by Method 2 on a PEPE | 202 |
| 4.41a | Method 1 for Transposing a Triangular Matrix on the PEPE | 207 |
| 4.41b | Method 2 for Transposing a Triangular Matrix on the PEPE | 207 |
| 4.42 | Parallel Adds of 13 Elements | 224 |
| 4.43 | General Covariance Computation Data Structure for STARAN-Type Memory | 230 |
| 4.44 | Procedure for Covariance Computation | 231 |
| 4.45 | Covariance Computation Data Structure for STARAN-Type Memory | 233 |
| 4.46 | Procedure for Covariance Computation | 234 |
| 4.47 | Covariance Computation Data Structure for STARAN-Type Memory | 235 |
| 4.48 | Procedure for Covariance Computation | 236 |
| 4.49 | Covariance Computation Data Structure for STARAN-Type Memory | 238 |
| 4.50 | Covariance Computation Data Structure for STARAN | 239 |
| 4.51 | Timing Figures for Covariance | 240 |
| 4.52 | Algorithm for GJ with K Steering Vectors | 244 |
| 4.53 | Algorithm for GJ with K Steering Vectors Transposed in the Array Memory | 245 |
| 4.54 | Program for GJ with K Steering Vectors Transposed in the Array with 9K Bit Word Size where N < 270 | 246 |

## LIST OF TABLES


| Number | | Page |
|---|---|---|
| 4.1 | Real Operation Counts for Sequential Processor Implementations for Determining Adaptive Weights | 66 |
| 4.2 | CDC 7600 Instruction Timings | 77 |
| 4.3 | CDC 7600 CPU Timings in Milliseconds | 82 |
| 4.4 | Logarithmic Timing Predictions for the CDC 7600 | 89 |
| 4.5 | Values of $t_{op}(X)/t_{su}(X)$ (and Their Reciprocals) | 97 |
| 4.6a | Number of Real Operations for 1 Iteration of IMU | 114 |
| 4.6b | Number of Real Operations for IMU to Form K Sets of Weights from $N_s$ Samples | 114 |
| 4.7a | Real Operation Counts as Function of Sample Size = $N_s$ for Implementation 1 for Calculating the Sample Covariance Matrix on a Vector Pipeline Processor | 118 |
| 4.7b | Real Operation as a Function of Sample Size = $N_s$ for Implementation 2 for Calculating the Sample Covariance Matrix on a Vector Pipeline Processor | 118 |
| 4.7c | Real Operation Counts as a Function of Sample Size = $N_s$ for Implementation 3 for Calculating the Sample Covariance Matrix on a Vector Pipeline Processor | 118 |
| 4.8a | Real Operation Counts for GJ | 127 |
| 4.8b | Real Operation Counts for Decomposition | 127 |
| 4.8c | Real Operation Counts for Decomposition with Augmentation | 128 |
| 4.8d | Real Operation Counts for Decomposition with Identity Matrix Augmentation | 130 |
| 4.8e | Real Operation Counts for First Back Substitutions | 132 |
| 4.8f | Real Operation Counts for Second Back Substitutions | 133 |
| 4.9 | Dominating Terms of Real Operation Counts; $\binom{m}{n}$ Denotes "either m or n" Depending on Algorithm, SUM Counts are Included in ± Counts | 134 |

LIST OF TABLES (Cont'd)


| Number | | Page |
|---|---|---|
| 4.10 | 64-Bit Performance Improvements of the CDC STAR 100-A over the CDC STAR-100 | 146 |
| 4.11 | CDC STAR-100 CPU Timings in Milliseconds | 148 |
| 4.12 | Logarithmic Timing Predictions for the CDC STAR-100 | 153 |
| 4.13 | Characteristics of the CRAY-1 Computer System | 164 |
| 4.14 | CRAY-1 Timings for Various Scalar and Vector Operations | 170 |
| 4.15 | CRAY-1 CPU Times in Milliseconds | 175 |
| 4.16 | Logarithmic Timing Predictions for the CRAY-1 | 182 |
| 4.17 | Operation Timings for the PEPE | 193 |
| 4.18 | Real Operation Counts for Updating the Sample Covariance Matrix on the PEPE | 208 |
| 4.19a | Real Operation Counts for GJ on the PEPE | 211 |
| 4.19b | Real Operation Counts for Unaugmented Decomposition on the PEPE | 212 |
| 4.19c | Real Operation Counts for Augmented Decomposition on the PEPE | 212 |
| 4.19d | Real Operation Counts for the First Back Substitution on the PEPE | 213 |
| 4.19e | Real Operation Counts for the Second Back Substitution on the PEPE | 213 |
| 4.20a | Possible Implementations (Augmented) for Solving MW = $\overline{S}$ on the PEPE | 216 |
| 4.20b | Possible Implementations (Unaugmented) for Solving MW = $\overline{S}$ on the PEPE | 216 |
| 4.21a | Total Real Operation Counts for Implementations for Solving MW = $\overline{S}$ on the PEPE | 218 |
| 4.21b | Total Real Operation Counts for Implementations for Solving MW = $\overline{S}$ on the PEPE When K=1 | 218 |

## EVALUATION

The significance of this contractual effort is the realization of generating a large number of adaptive weights over more than one domain in radar systems. Such efforts will eventually render fully adaptive radar systems feasible. It supports RADC Technology Plan TPO R4B by providing a technical basis for adaptive radar operation in a hostile and interferring electromagnetic environment. The results derived herein will serve as a basis for using modern computer architecture to solve the large array problem.

VINCENT VANNICOLA
Project Engineer

# 1.0 INTRODUCTION

## 1.1 STATEMENT OF THE PROBLEM

The objective of this study is to determine the effect of computer system architectures (particularly parallel and associative processing systems) on selecting adaptive algorithms for the signal-to-noise ratio (SNR) optimization of a multichannel input radar system with a large number of weights. The radar system to be considered has channel inputs in all three signal domains--spatial, temporal, and polarization--with multiple inputs in each domain. The adaptive algorithms evaluated must sense the environment and adjust at least 200 complex weights imposed on the array elements, on the delay taps off each element, and on the polarization sensors so the signal-to-noise ratio is optimized. This sensing must be accomplished on a time-varying basis, as the environment dictates, with minimum convergence time, minimum noise, and maximum stability. There has to date been a question about the feasibility of treating this number of weights computationally with any algorithm or computer architecture.

In order to calculate at least 200 complex weights quickly enough to adapt to the changing environment, we must not only choose a fast adaptive algorithm and a fast computer, but the algorithm must be well suited to the particular computer on which it is to be implemented. By considering vector pipeline, associative, parallel, multiprocessor, and sequential architectures, this report examines the subtle interaction between computer architecture design and adaptive algorithm selection.

## 1.2 RESTATEMENT OF THE PROBLEM IN MATHEMATICAL TERMS

Each antenna element in an adaptive array produces an input signal, which may be written

$$X_j(t) = N_j(t) + S_j(t),\ j=1,\ \ldots,\ N$$

where X is the complex waveform as a function of time t, $N$ is the noise component, S is the signal component, the subscript j designates the particular antenna element, and N is the total number of antenna elements. X, $N$, and S may be thought of as complex column vectors of their components:

$$X(t) = \begin{bmatrix} X_1(t) \\ X_2(t) \\ \vdots \\ X_n(t) \end{bmatrix} .$$

Instead of these continuous waveforms, we will usually deal with the sample vectors $X^{(i)}$, $N^{(i)}$, and $S^{(i)}$, which are defined by:

$$X^{(i)} = X(t_i)$$

where the $t_i$ are closely spaced points in time. Let W be a column vector of complex numbers. We form the filter function

$$F = W^T X$$

as the dot-product of W and the input vector X.

Our problem is to choose the weights W to optimize the SNR of the system, in order to maximize the probability of detection of the signal in the presence of jammers and clutter. These weights are a function of the input vectors X(t), or rather, the sequence of sample input vectors $X^{(i)}$. This report not only explores different algorithms for

calculating the weights W and filter function F, but also the effects of different computer architectures on the implementations of these algorithms. These effects include execution speed, accuracy, convergence rates, cost of the computer, and fault tolerance.

## 1.3 APPROACH

We first examined the operating environment of the radar and made assumptions regarding it. This environment includes the types of jammers and clutter the radar is likely to encounter, near-field effects, errors in array-element excitation, and the number of degrees of freedom. We made a minimum of limiting assumptions in order that our system be as generally applicable as possible.

We then examined the effects of the operating environment and assumptions on the mathematical formulation of the problem and made further assumptions. We translated, for example, conditions on the input voltages into conditions on their sample covariance matrix. As before, we tried to maintain the utmost generality.

We then chose a set of algorithms satisfying these assumptions, such as loops and sample covariance matrix inversion. In order to limit our field of choice we set a constraint on the minimum convergence rate of the algorithms (i.e., the number of sample vectors that are required for the SNR to approach the optimum). It has been shown that, for the sample covariance matrix inversic method, it is theoretically possible to get within 3 dB of optimum with 2N samples (where N is the number of weights) under certain assumptions regarding the noise distribution [Reed, Mallett, and Brennan, 1974]. Since using the actual covariance matrix (which of course is not available) instead of the sample

matrix provides the optimal weights (see Part 2 of this report), this theoretical result was chosen as a convergence criterion for all algorithms. Since simulations have shown control loops do not converge this quickly in general [Reed, Mallett, and Brennan, 1974], they were eliminated from further consideration. This convergence-rate test was also the only requirement we made regarding algorithm accuracy, as we assumed the SNR achieved is a good measure of this accuracy.

We then examined the remaining algorithms and all their variations with respect to their mathematical properties, particularly their computational complexity, i.e., how fast they could be executed on a computer. Algorithm speed is computer-dependent, so each algorithm had to be considered in conjunction with each possible computer architecture on which it could be implemented. This divided the study into an algorithm section and an implementation section. Most of the algorithms depend on solving a set of simultaneous linear equations, and we surveyed a great deal of literature on this problem, dealing with both algorithms alone and specific implementations. Poor implementations were eliminated, and the most promising ones were coded and run. Both simulations to test convergence and timing runs were made and the results compared with theoretical predictions. The cost and fault tolerance of the computers were also considered.

Finally, synthesizing the knowledge we gained in the above analysis, we discussed a parallel architecture designed specifically for our problem and made to take advantage of as much inherent parallelism in the most practical algorithm we found. In choosing this architecture, we took into account

computer cost, fault tolerance, ease of manufacturing the hardware, and the relative amounts of inherent parallelism and computational complexities in the algorithms we studied.

This report presents not only the final numerical comparisons obtained, but also the basic problems, tradeoffs, and hidden complexities of the problem.

## 1.4 CONCLUSIONS AND SUMMARY

In order to fully explore all possible solutions to the overall problem of forming adaptive weights, we have had to consider not only all algorithms and all computer architectures, but every combination of algorithm and architecture (i.e., implementation). The number of possible implementations is far greater than the number of either algorithms or architectures (being the product of the two), but we nevertheless have attempted in this report to deal with these implementations comprehensively. This comprehensive approach is demanded because our results have shown conclusively that it is impossible to effectively select the optimal implementation without considering the totality of all possible implementations. This statement is valid not just for the problem of forming adaptive weights, but for any problem involving parallel and associative architectures. The current literature on this problem has <u>not</u> taken this approach; it has either dealt solely with algorithms and not the difficulties of implementing them or has tried to deal with isolated portions of certain implementations, often ignoring the computer-time-consuming problems of data movement and system overhead. Such limited approaches cannot be used to design a successful real-time system.

Our most important conclusion is that the sample covariance matrix algorithms can be successfully implemented to form adaptive weights for a 200-weight system, and the computational results agree with the theoretical prediction of achieving an SNR within 3 dB of optimum after 2N samples, where N is the number of weights [Reed, Mallett, and Brennan, 1974]. This result also implies that the implementation will work for any number of weights less than 200. A plot summarizing this result is shown in Figure 1.1.

We also compare the performance of the different machines we studied. Figure 1.2 presents log-log plots of times in milliseconds versus the number of weights for the CRAY-1, CDC 7600, CDC STAR-100, and STARAN. The times shown for the first three machines are the total times actually required to form a sample covariance matrix from 2N samples (where N is the number of weights) and then to solve for the adaptive weights, using various algorithmic varieties of the sample covariance matrix inverse method and the particular implementations we decided were optimal for each machine. These times are upper bounds for the results obtainable in a real system, since as few system-specific assumptions which could increase program efficiency were made as possible. The STARAN times, on the other hand, are a theoretical lower bound for the time it takes to perform the decomposition part of the algorithm only. This lower bound was obtained by assuming that the STARAN-E configuration was available with sufficient memory available to contain the data in such a way that N complex floating-point multiplications and additions could be performed with sufficient parallelism that they would only require the time of one single-precision real floating-point multiplication and addition, 240 microseconds ["Comparison of the Basic STARAN Architecture

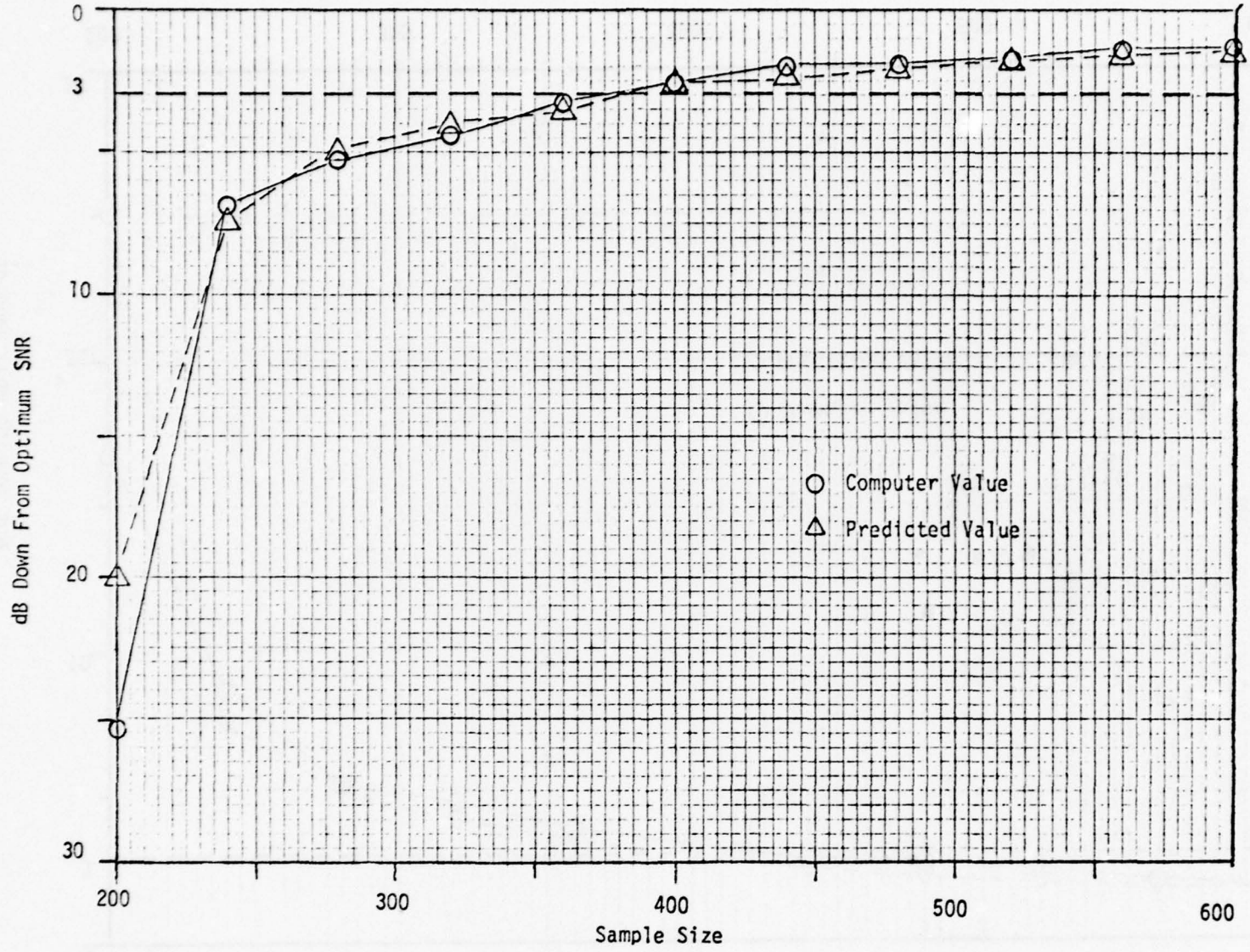



Figure 1.1a Results of Radar Simulation with One Jammer for 200-Weight Adaptive Array Using the Sample Covariance Matrix Inverse Method

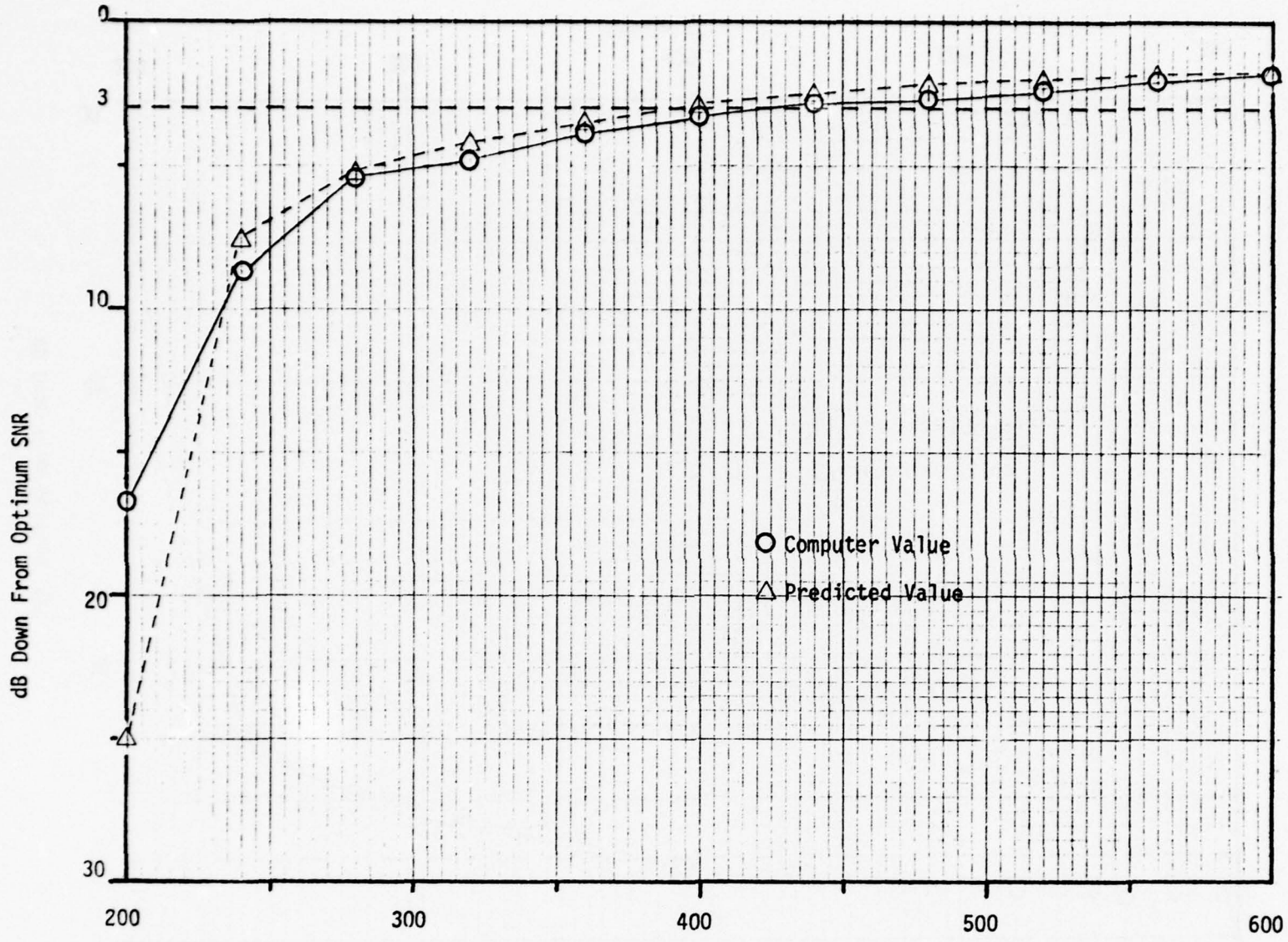



Figure 1.1b Results of Radar Simulation with Two Jammers for 200-Weight Adaptive Array Using the Sample Covariance Matrix Inverse Method

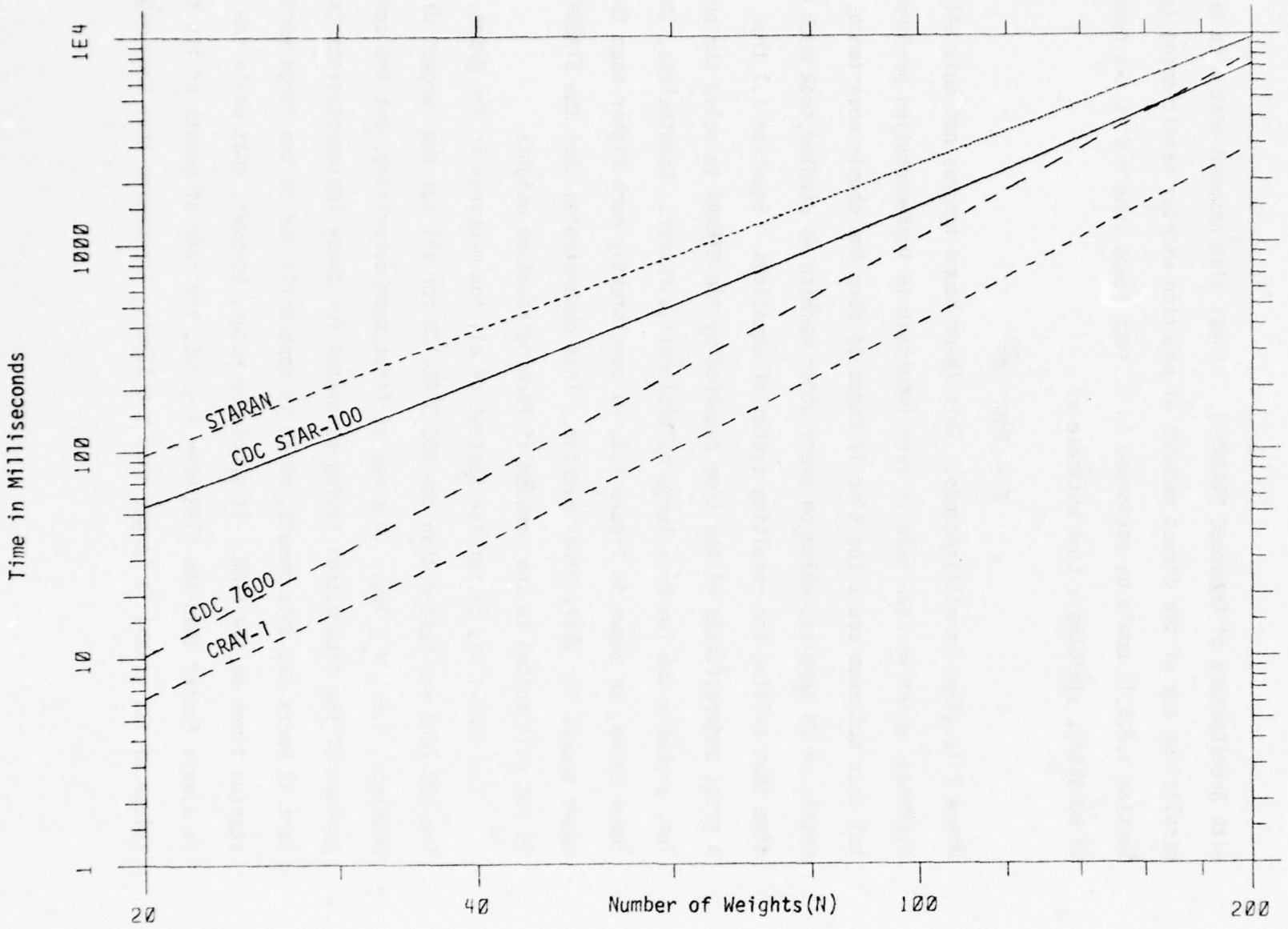



Figure 1.2 Implementation Timing Comparisons (for 2N Samples)

with Architecture of Enhanced STARAN"]. It was also assumed that with this parallelism any of the direct methods of solution without partitioning (see Section 3.2.5.1) could be performed in $N^2$ such steps (where N is the number of weights), yielding a time estimate of

$$T = .240 \cdot N^2 \qquad 1.1$$

where T is given in milliseconds. This lower bound ignores not only all overhead, which we have said is very important in implementation performance, but does not even count the time it takes to form the sample covariance matrix of 2N samples, which, on every other machine we studied, took much more time than solving the resulting system of equations. Equation 1.1 then is a gross underestimate of the time required by the STARAN to solve the problem, probably too low by a large multiplicative factor. Nonetheless, this lower bound, as shown in Figure 1.2, is consistently much higher than the upper bounds for every other machine. This demonstrates that the STARAN is not well suited to the problem of forming adaptive weights.

The CRAY-1 was by far the fastest of all the machines in the graph. The CDC 7600 was faster than the CDC STAR-100 for all but the largest dimensions, i.e., $N \gtrsim 160$. This may at first seem surprising, but the comparison of the theoretical timing estimates for these implementations (see Part 4) bears out this result, which is apparently due to the large vector startup times on the STAR. It should be noted, however, that while the CRAY is always faster than the STAR when $N \leq 200$, the rate of growth of its time (slope of its graph) is larger than the STAR's, indicating that they might

cross over at some point N > 200. This implies that for larger dimensions it would be necessary to compare the two machines again to determine which is fastest.

The question "Which implementation is best?" has no simple answer, then, but is a function of many radar engineering considerations, such as the number of weights and the speed with which they must be updated (see Section 2.3). For example, to form just 30 complex weights in under 15 milliseconds using available technology would require at least one CRAY-1, a large investment. Other current machines with vector and sequential architectures may be suitable for smaller problems. U.S. Army SOTAS (Stand Off Target Acquisition System) Emulator, designed and developed by TSC, is a real-time adaptive radar system with 5 weights, with the weight calculation being done using the sample covariance matrix method implemented on a sequential mini-computer.

There are several conclusions of this report which apply to any algorithm or architecture to be considered as a solution to the problem of forming adaptive weights:

1) It is necessary to remember the difference between algorithm and implementation, since the traditional methods of ranking algorithms (such as operation counts) will not always correctly rank their corresponding implementations. Choosing the best implementation, not algorithm, is what is important, since it is the implementation that actually provides the solution to the problem. Operation counts are good, however, at providing upper bounds on implementation performance.

2) Any implementation may be improved by using separate machines for the sample covariance matrix calculation and solving the set of simultaneous linear equations for the weights and running these machines in parallel. This approach will not decrease the time between receiving the samples and forming the weights they determine, but it will allow the weights to be updated more frequently. This approach also allows for greater implementation optimization, since there is no reason the two machines even have to have the same architecture, if the two facets of the solution demand different ones.

3) Since all the algorithms considered involve complex arithmetic, it would be a great advantage in speed if a computer had hardware parallel complex arithmetic instructions. As discussed in the section on complex arithmetic, it is possible to perform a complex multiply in the time it takes to do one real multiply (four in parallel) and one real addition (two in parallel) whereas on a sequential processor it might take the time of four real multiplies and two real additions. This function can be implemented in the hardware as part of any architecture and should be transparent to the user.

4) Because of the highly arithmetic nature of the problem, we have concluded that associativity cannot be used, except to enable processors in a parallel machine.

In this report, we have demonstrated the computational feasibility of solving for up to 200 complex adaptive weights and have provided a needed reference document for radar design engineers confronted by this difficult problem.

# 2.0 RADAR CONSIDERATIONS

## 2.1 RADAR OPERATING ENVIRONMENT

Adaptive arrays are of current interest for a variety of different radar applications. An adaptive system senses the external noise environment and adaptively controls the array antenna pattern to optimize the performance of the system. The types of external noise which are sensed and rejected by an adaptive radar include jamming, unintentional interference from friendly emitters, and backscattering from clutter. While it is possible in theory to design antennas with very low sidelobe levels which would discriminate against these types of sidelobe interference, the sidelobe levels which are achieved in practice are limited by errors in array-element excitation, radome reflection, and multipath scattering from objects near the radar. An adaptive receiving array senses each of these effects and readjusts the element weights for optimum performance. As a result of these effects, in addition to the possibility of unequally spaced or dissimilar antenna elements, no assumptions can be made about the signals produced from adjacent or close antenna elements; in particular, one cannot assume that the signals differ simply in phase.

The types of clutter backscattering of importance in various radar systems include backscattering from the terrain, rain clutter in higher-frequency systems, auroral clutter in HF systems, echoes from chaff, and possible echoes from nuclear effects. An adaptive system can sense each of these types of external noise and include their effects in the antenna pattern optimization.

The application of adaptive array technology is of current importance in systems with frequencies ranging from HF to X-band. An important

limitation of HF over-the-horizon radars results from auroral backscattering which is Doppler shifted by auroral motion. This results in echoes which cannot be distinguished from moving-target echoes by Doppler filtering. Adaptive control of an array antenna pattern provides a method of rejecting this type of interference. At the other end of the commonly used frequency band, X-band, sidelobe cancellation of jammers is the primary application of adaptive array technology. The multiple channel sidelobe canceller is a special case of an adaptive array. A system with N auxiliary elements can adaptively null up to N jammers.

At higher frequencies, an array contains a large number of individual antenna elements. For example, an S-band array of $10m^2$ area contains 3,000 half-wavelength spaced elements. In these higher-frequency systems, sub-array outputs can be formed using pre-computed weights and the sub-array outputs added with adaptively controlled weights. Sub-arraying reduces the number of adaptive degrees of freedom and simplifies the adaptive processor. Different sub-array configurations can be used, e.g., column outputs added adaptively to provide azimuth pattern control or row outputs adaptively summed for control in elevation. Individual element outputs can be used in a multiple channel sidelobe canceller configuration. In the interests of generality, we will assume that all subarraying has been done before we begin processing the data.

It is also advantageous in many systems to control the frequency response of the system adaptively. Adaptive filtering is closely analogous to adaptive array processing and is of current interest in some MTI radars. For wide bandwidth jammer cancellation, it is advantageous to use multiple outputs from each element of the array. These outputs are separated in

time, e.g., obtained from tapped delay lines behind each element. In a digital system, consecutive samples of each element output provide the required set of time-delayed samples. A system which is adaptive in both time and angle, where a separate adaptively controlled weight is applied to each of K time samples from each of N elements by controlling KN weights, provides excellent wide bandwidth jammer cancellation. This technique of array-frequency control is important in systems where multipath echoes contribute significantly to the external noise input.

The performance of an adaptive radar will generally depend upon the number of degrees of freedom of the system. Additional degrees of freedom improve the capability to reject multiple interference sources, interference of both polarizations, multipath, wideband interference, and clutter returns which are continuously distributed in angle. This study addresses the problem of adaptive weight computation in a system with a large number of degrees of freedom. The method of weight computation considered in this study, based on the sample covariance matrix, provides good performance independent of the distribution of eigenvalues of the covariance matrix. The adaptive technique considered here is applicable to antenna array processing, to adaptive filtering, and to systems which are adaptive in both angle and frequency.

## 2.2 THEORY OF ADAPTIVE RADAR

Adaptive array processors are currently of interest for a variety of applications in the radar, communication, and sonar fields. In an adaptive processor, a separate coherent output is obtained from each element (or

subarray) of a phased-array receiving antenna. The output of each element channel is sampled and multiplied by a complex weight, i.e., adjusted in both amplitude and phase, and these weighted outputs are added coherently to form Doppler-compensated receiving beams. The complex weights of this space-time filter are controlled by adaptive loops. By this means, the illumination function and the Doppler reject filters of the receiving array antenna are controlled adaptively.

Such adaptive systems are designed to continuously maximize the probability of detection for a fixed false-alarm rate. This maximization can be shown to be equivalent to a maximization of a generalized signal-to-noise ratio [Brennan and Reed, 1973]. The adaptive processor senses both the receiver noise in the individual space-time channels and the external noise field in performing this optimization. External noise may include both discrete sources in the sidelobe region (e.g., interfering signals in a communication system or jammers in a radar system) and continuously distributed noise (e.g., sky noise in radio astronomy or clutter in a radar system).

Let a radar, possibly moving, emit an electromagnetic signal. Suppose an echo from a target at a given range delay is received by a set of N sensors. Let each sensor yield K samples of the received signal in time. Denote the expected sensor time-sampled data from the target by the column vector

$$S = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_n \end{pmatrix} \quad . \qquad 2.1$$

Where $S_k$ is a complex number, the sampled phase and envelope for k = 1,2,..., n, and n=N·K is the total number of space-time samples.

Vector S in Eq.2.1 is called the signal vector. If the target is moving with respect to the radar, S would contain the expected relative Doppler and polarization phase factors, both with regard to the sensor's special position and its sampling time.

If vector S is "tuned" to a moving target at position R, returns from stationary targets at other positions act as clutter or noise to the detection process. Other noise components include receiver noise, interference, sky noise, etc. Let the column vector

$$N = \begin{pmatrix} N_1 \\ N_2 \\ \vdots \\ N_n \end{pmatrix} \qquad 2.2$$

denote the total return for noise alone, i.e., the return assuming there was no target at the position R of interest. The sum of vectors S and $N$ is the signal-plus-noise vector

$$X = S + N. \qquad 2.3$$

The radar problem is to detect the presence of signal S in noise $N$.

To detect S in Eq. 2.3, one puts the received vector through a "filter" with weights,

$$W = \begin{pmatrix} W_1 \\ W_2 \\ \vdots \\ W_n \end{pmatrix} . \qquad 2.4$$

The output of filter W is the scalar

$$F = \sum_{k=1}^{n} W_k X_k = W^T X \qquad 2.5$$

where T denotes matrix transpose. The expected value of F is zero for noise alone and

$$EF = \sum_{k=1}^{n} W_k EX_k = \sum_{k=1}^{n} W_k S_k = W^T S \qquad 2.6$$

for signal plus noise. Similarly, the noise power or variance of F is

$$\sigma^2 = E|F|^2 - |EF|^2 = \bar{W}^T E\bar{N}N^T W = \bar{W}^T M W \qquad 2.7$$

where the bar denotes complex conjugation and M is the covariance matrix of the noise process, i.e.,

$$M = E\bar{N}N^T = (E\bar{N}_j N_k) \qquad 2.8$$

Using these definitions, the following theorem can be proven [Brennan and Reed, 1973]:

Theorem 1: Assume a radar (or sonar) transmits a waveform and receives n space-time samples. For noise alone, say, hypothesis $H_0$, the receiver observes the vector $X = N$ where $N$ is the n component column vector defined by Eq. 2.2. For signal plus noise, say, hypothesis $H_1$, the receiver observes the vector $X = S + N$, where S is the n-component column vector defined by Eq. 2.1. Let W be the filter vector, the n-component

column vector defined by Eq. 2.1. Let W be the filter vector, the n-component column vector of weights defined by Eq. 2.4. Then the response of the filter to the received observables is

$$F = \sum_{k=1}^{n} W_k X_k = W^T X \quad .$$

Finally, suppose that the components of $N$ are jointly distributed Gaussian variates, and that the covariance matrix is $M = E\bar{N}N^T$.

With the above definitions and assumptions, the filter which gives the maximum probability of detection $P_D$ for a fixed probability of false alarm $P_F$ is

$$W = kM^{-1}\bar{S} \qquad 2.9$$

where k is a nonzero complex number. The probability of detection for this optimum filter is

$$P_D = Q\left(\sqrt{\bar{S}^T M^{-1} S}\ ,\ \sqrt{2 \log \frac{1}{P_F}}\right) \qquad 2.10$$

in terms of the false-alarm probability $P_F$, where $Q(\alpha,\beta)$ is the Q function defined by

$$Q(\alpha,\beta) = \int_{\beta}^{\infty} \left[\frac{-(\nu^2 + \alpha^2)}{2}\right] I_o(\alpha\nu)\, d\nu \quad . \qquad 2.11$$

In practice, of course, the actual covariance matrix M is not available. Instead, we use the sample covariance matrix, which we will denote here by $\hat{M}$. Rewrite the samples $X_1,\ldots,X_n$ as $X_1^{(1)},\ldots,X_N^{(1)},X_1^{(2)},\ldots,X_N^{(2)},\ldots,X_1^{(K)},\ldots,X_N^{(K)}$, grouping together samples taken at the same moment in time (denoted by identical superscripts) and ordering them in these groups by antenna element number (denoted by the subscript). $\hat{M}$ may be estimated in several ways from the samples $X_1^{(1)},\ldots,X_n^{(k)}$ (see Section 3.2.4), but the method used in the remainder of this report is

$$\hat{M}_{ij} = \sum_{k=1}^{K} \overline{X}_i^{(k)} X_j^{(k)} \quad . \qquad 2.12$$

Let $p(\hat{M})$ be the normalized signal-to-noise ratio: SNR $(\hat{M})$/SNR(M), where SNR(M) is the optimal SNR (obtained from using the true matrix, M, in forming the weights) and SNR $(\hat{M})$ is the actual SNR obtained using the sample matrix, $\hat{M}$. After making certain assumptions about the distribution of $p(\hat{M})$, it can be shown that the expected value of $p(\hat{M})$ is [Reed, Mallett, and Brennan, 1974]

$$E(p(\hat{M})) = (K + 2 - N)/(K + 1) \quad . \qquad 2.13$$

This is the expected loss-in-power ratio if only K samples of data were used to estimate $\hat{M}$ in Eq. 2.12. Expressed in decibels, this expected loss is

$$\text{loss} = -10 \log_{10} \{E(p(\hat{M}))\} = -10 \log_{10} [(K + 2 - N)/(K + 1)] \quad . \qquad 2.14$$

Hence if one wishes to maintain an average loss ratio of better than one-half (less than 3 dB), at least $K = 2N - 3 \approx 2N$ samples are needed. This result has been verified by simulations [Reed, Mallett, and Brennan, 1974].

## 2.3 RADAR ENGINEERING CONSIDERATIONS

We will now discuss several important radar design questions which must be answered before the optimal implementation can be selected to form a set of adaptive weights. The major design consideration is the number of weights, N. The running times of all implementations are functions of N, usually second- or third-order polynomials. The number of weights is determined by the radar environment, including such factors as the number and types of jammers, clutter environment, and beamforming (in fully adaptive systems this includes main beam width, sidelobe levels, etc.).

Another major consideration is time. Two time factors are important: 1) the length of time the covariance matrix is a good estimate of the environment, and 2) the length of time a given set of weights is applicable. These times may be dynamic or constant, depending on the system. Some factors that may determine these times are the method of scan (mechanical or electrical), sampling rate, single- or multiple-beam mode, or system objective (e.g., OTH).

Other considerations affecting the time required to perform the weight calculation include the number of bits in the analog-to-digital converters (ADCs), the number of bits required to represent the weights, and special forms of the steering vectors.

A fourth group of system considerations, as in all systems, is costs, power consumption, maintainability and reliability.

It is not possible to list all design criteria for adaptive antennas and sidelobe cancellers. We feel that the designer of the weight calculation subsystem should be involved with the entire system design and be able to impact that design. For example, in order to meet speed requirements and cost constraints, the designer may want to limit either the number of weights, the number of bits in the ADC, or the desired SNR.

## 2.4 A SAMPLE VOLTAGE VECTOR MODEL

In order to determine if the sample covariance matrix approach is feasible when N, the number of weights, is at least 200, we had to use a model of the sample voltage vectors to write a computer simulation to test the algorithm and its implementation.

The radar test problem is arranged so that the interferers can be specified by their eigenvalues. To do this simply, a linear antenna array with uniform spacing and weighting was chosen. With this configuration it is easy to form multiple beams and place an interferer in each beam so that each beam output contains only power from that interferer. Since each beam output is then independent, the covariance matrix of the beam output is diagonal and the interferer powers are the eigenvalues. It is easy to transform the problem to element space, using a unitary transformation.

To implement this approach, the interferers are placed so that all except one are at nulls of the beam pattern for each beam. The voltage output of a beam formed by summing all element outputs is given by

$$X_b = [1 + e^{i\psi} \ \ldots \ e^{i(N-1)\psi}] = \frac{\sin N\,\psi/2}{\sin \psi/2} \qquad 2.15$$

where $\psi = 2\pi D \sin\theta$ ,

$\theta$ is an angle measured from the line of the antenna to a point source

D is the antenna length in wavelengths.

If interferer positions are chosen so that $\sin(N(\psi_n-\psi_m)/2) = 0$ for $n \neq m$, they satisfy the condition for independence. This will be true when $\frac{N}{2}(\psi_n-\psi_m) = k\pi$ where k is an integer. The desired interferer positions therefore are

$$\psi_k = \frac{2\pi k}{N} \qquad k=1,\ldots, N-1 \quad . \tag{2.16}$$

In element space the voltage at each of N elements is given by

$$X_n = \sum_{k=1}^{K} R_k \sqrt{\lambda_k}\, e^{i2\pi k(n-1)} + \sqrt{Q_n} \cdot R_n \tag{2.17}$$

for the K = N-1 interferers. The $\lambda_k$ are the powers in each interferer and the R's are independent zero-mean random variables with $|\overline{R_k}|^2 = 1$. The $R_k$ are included, when needed, to simulate the interferer variations between samples, and receiver noise whose power is $Q_n$.

Since the interferer positions are known, the true covariance matrix can be computed as

$$M_{m,n} = \overline{X^* X}_{m\ n} = \frac{1}{K}\sum_{k=1}^{K} \lambda_k\, e^{-i\,\frac{2\pi k}{N}(m-n)} \quad . \tag{2.18}$$

A stochastic sample covariance matrix using S samples can be obtained by forming

$$\hat{M}_{m,n} = \frac{1}{S} \sum_{s=1}^{S} {}_{s}X_{m}^{*} \, {}_{s}X_{n} \qquad 2.19$$

where the voltages are obtained from Equation 2.17.

With this model the eigenvalues $\lambda_k$ can be chosen to span any desired range of values, and the number of interferers can be varied up to N-1 while specifying the eigenvalues.

# 3.0 MATHEMATICAL TECHNIQUES

## 3.1 STATEMENT OF THE PROBLEM, NOTATION, APPROACH, AND ASSUMPTIONS

The problem is to optimize the SNR of an adaptive system, where X denotes the complex column vector of random variables describing the receiver inputs (length = N), and F, the desired filter function. As discussed above, if we write X as

$$X = N + S \quad , \tag{3.1}$$

where S is the column vector of random variables describing the signed vector and $N$ the column vector of random variables describing the noise vector, then the filter function, F, is given by

$$F = W^T X \quad , \tag{3.2}$$

where $W^T$ is the transpose of a complex column vector of weights, W, which satisfies

$$qMW = \overline{S} \quad , \tag{3.3}$$

where q is an arbitrary constant (nominally 1). $\overline{S}$ denotes the complex conjugate of the steering vector, S (a column vector), and M is the covariance of the noise vector, $N$,

$$M = E\,(N^* N) \quad , \tag{3.4}$$

where E $(\cdot)$ denotes expectation and $N^*$ is the conjugate transpose of $N$

$$N^* = (\overline{N})^T = (\overline{N^T}) \quad . \tag{3.5}$$

Our approach is to calculate the weights, W, and then calculate F by the inner (dot) product in Equation 3.2. The inputs we have to work with are sample voltage vectors $X^{(i)}$ sampled periodically in time, and their length. There are several algorithms available to perform this task.

Our criteria for determining the best algorithm will be twofold. First, the convergence rate: How many samples are required to get within a certain threshold (such as 3 dB) of the optimum threshold (if the algorithm can get there at all)? Once we have chosen the algorithms that require the fewest samples, we will pick the one that requires the least time to perform. Computational accuracy and numerical stability will be criteria only insofar as they affect the convergence rate.

We assume our sampling rate for the $X^{(i)}$'s is high enough with respect to the speed of change of the distribution of the noise process that we may extract from the $X^{(i)}$'s information about the instantaneous covariance matrix, M.

Of the two general classes of algorithms, control loops and estimating the matrix M (or its inverse) and solving Equation 3.3, we eliminate the loops on the basis of the first criterion above--its slow convergence rate (this has been discussed above).

We must make a further assumption: solving Equation 3.3 presumes the existence of $M^{-1}$. Since M is a complex covariance matrix, we know several things about it a priori:

1) it is Hermitian; i.e., $M_{ij} = \overline{M}_{ji}$ since $M_{ij} = E(\overline{N}_i N_j)$
$= E(\overline{N_i N_j}) = E(\overline{N_j N_i}) = \overline{M}_{ji}$ ; and

2) it is positive semi-definite; i.e. $Q^*MQ \geq 0$ for arbitrary Q, since

$$Q^*MQ = \sum_{j=1}^{n} \overline{Q_j} \left( \sum_{k=1}^{n} M_{jk} Q_k \right)$$

$$= \sum_{j=1}^{n} \sum_{k=1}^{n} \overline{Q}_j \, E(\overline{N}_j N_k) Q_k = \sum_{j=1}^{n} \sum_{k=1}^{n} E(\overline{Q_j N_j} Q_k N_k)$$

$$= E\left( \left( \sum_{j=1}^{n} \overline{Q_j N_j} \right) \left( \sum_{k=1}^{n} Q_k N_k \right) \right) = E(\overline{R}R) \geq 0, \text{ where}$$

$$R = \sum_{j=1}^{n} Q_j \overline{N}_j \text{ and } \overline{R}R \geq 0 \text{ for any R.}$$

If M is Hermitian positive semi-definite, then it will be nonsingular (i.e. $M^{-1}$ will exist) if and only if it is positive definite, i.e., $Q^*MQ > 0$ for all arbitrary non-zero Q. This is because $Q^*MQ = E(\overline{R}R) = 0$ if and only if R is identically 0 for a non-zero Q, which is true if and only if the $N_j$ are linearly dependent, which is true if and only if M is singular. Hence, we assume M is positive definite, an eminently reasonable assumption since the necessary existence of receiver noise, which is uncorrelated, prevents the exact linear dependence of the $N_j$'s.

In Part 2, the discussion of the radar operating environment, we stated that no assumptions could be made regarding relationships of input voltages from adjacent or close antenna elements. The implication is that no assumption can be made about the form of M. In particular,

since we cannot assume that signals differ in phase only, we cannot assume that the matrix M is Toeplitz (i.e., $M_{ij}$ is a function of i-j only).

We make no assumptions about the steering vector $S_j$; it may be arbitrary. If, in practice, it has a special form (like a unit vector as in sidelobe cancelers), then the algorithms we discuss may be changed to take advantage of this knowledge. Also, since there can be only N independent beams formed by a system of N weights, and hence only N independent steering vectors (or in mathematical terms, a group of independent vectors of dimension N can have at most N members), we will assume that we will not want to calculate the weights corresponding to more than N steering vectors for one sample covariance matrix.

## 3.2 ALGORITHMS FOR SOLVING MW = $\bar{S}$

### 3.2.1 Introduction

The algorithms for solving MW = $\bar{S}$ may be divided into two classes: those which update an estimate of M with each sample vector (and subsequenty solve MW = $\bar{S}$ by several methods discussed below) and those which update an estimate of $M^{-1}$ (and subsequently multiply the inverse $M^{-1}$ by $\bar{S}$ since $W = M^{-1}\bar{S}$). There is only one basic algorithm in this second class, the inverse matrix update (IMU) method. In the first class, once M has been estimated, the algorithms may be further divided into direct and iterative methods. Direct methods are those which would, were infinite precision available, yield exact solutions in a finite predetermined number of steps. Iterative methods take an initial guess at the solution and produce a sequence of solutions converging, it is hoped, componentwise tc the true solution.

The direct methods may be broken into two classes, depending on whether they use partitioning of the matrix (divide and conquer approach) or operate only on whole rows and columns. It has been shown [Klyuyev and Kokovkin-Shcherbak, 1965] that Gaussian Elimination (GE) is optimal with respect to operation counts in this second subclass for a general matrix. This subclass also includes Gauss-Jordan Elimination (GJ) and Cholesky factorization, the latter of which is a modification of Gaussian Elimination which takes full advantage of the fact that a matrix, M, is positive definite Hermitian (both the actual matrix and its

estimate) and hence, when limiting oneself to full row and column operations, is optimal for the problem. Cholesky takes on two forms, one without square roots (abbreviated $LDL^*$, from the triangular factorization $M = LDL^*$ it provides) and one with square roots ($LL^*$). It should be noted that these are all $O(N^3)$ algorithms; i.e., the dominating term in their operation counts is an $N^3$ term. The partitioning algorithms, on the other hand, allow for algorithms of order $N^x$, where $x<3$. The smallest exponent obtained so far is $\log_2 7$, or approximately 2.81, by Strassen [1969]. His method involves partitioning each matrix and submatrix into 4 submatrices, and it has been shown by Hopcroft and Kerr [1971] that an $O(N^{\log_2 7})$ algorithm is the best obtainable with this partitioning scheme. It can be shown [Aho, Hopcroft, and Ullman, 1974] that if the matrix can be partitioned into $a^2$ submatrices of equal size, and a x a matrix multiplication can be performed with b noncommutative multiplications, then matrices can be multiplied with an $O(N^{\log_a b})$ algorithm. Since Bunch and Hopcraft [1974] have shown that matrix multiplication, matrix inversion, and triangular factorization are equivalent in computational complexity, this means that $MW = \bar{S}$ may be solved with an $O(N^{\log_a b})$ algorithm. Methods, however, of multiplying 3 x 3 matrices in 21 multiplications (resulting in an $O(N^{\log_3 21}) = O(N^{2.77})$ algorithm) or 4 x 4 matrices in 48 multiplications (resulting in an $O(N^{\log_4 48}) = O(N^{2.79})$ algorithm) have yet to be discovered. Partitioning methods often involve a great deal of overhead and have larger factors in front of their dominating terms than whole row

and column algorithms, so their asymptotic superiority often does not dominate until N is very large. Each direct method has two variations: solving for $M^{-1}$ and multiplying it by the steering vectors, and solving for the weights directly (with back substitutions or the equivalent).

The iterative methods we will consider are linear and of degree one; i.e.,

$$W^{(k+1)} = P^{(k)} W^{(f)} + Q^{(k)} \quad , \qquad 3.6$$

where $W^{(k)}$ is the $k^{th}$ iterative refinement of the solution of $MW = \overline{S}$. If P and Q are constant with respect to k, the iteration is called stationary, otherwise it is called nonstationary. We will discuss Point-Jacobi iteration, Gauss-Seidel iteration, successive over-relaxation (SOR) and chaotic iteration.

This hierarchy of techniques is presented in Figure 3.1. The "$M^{-1}$" and "Back Sub." at the bottom of the tree under the "Direct Methods" node indicate whether $M^{-1}$ will be calculated and multiplied by the steering vector $\overline{S}$ or if the weights will be calculated directly (by back substitution or the equivalent). These are labelled as sequential techniques to distinguish them from ones designed for or implemented on parallel processors. There is a certain amount of parallelism present in each algorithm of Figure 3.1 that may be exploited in an implementation on a parallel processor, but these algorithms (except chaotic iteration) were originally conceived in sequential terms and will be discussed in those terms. The question of exploiting parallelism for these methods is primarily left to

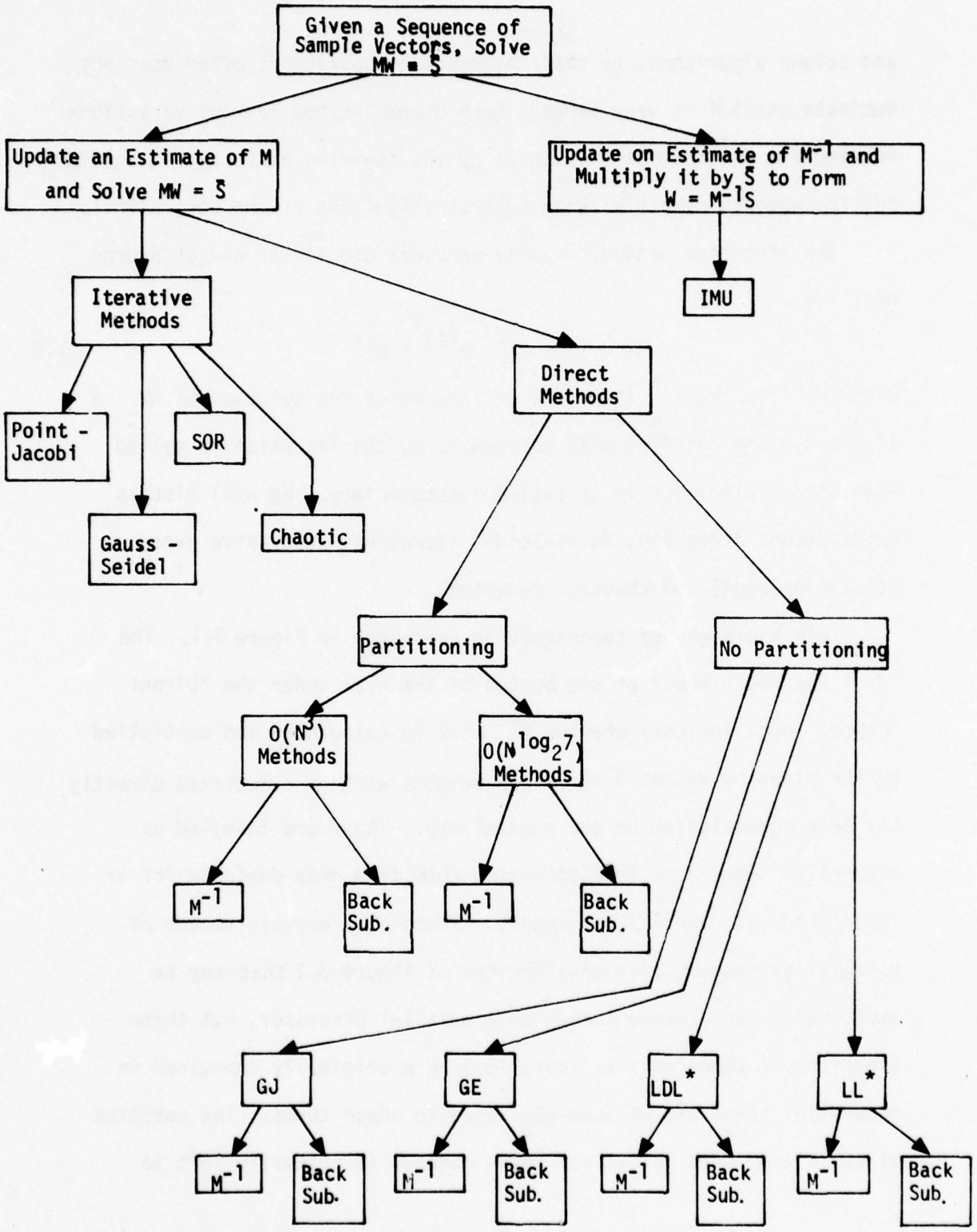



Figure 3.1 Summary of Sequential Mathematical Techniques

Part 4 on implementations.

There are algorithms, however, that were designed especially for parallel processors, in particular an algorithm which requires $N^5$ processors but can solve a system of equations in $O(\log^2 N)$ operations [Csanky, 1976]. These algorithms will be discussed separately.

Henceforth, N will denote the number of weights (dimension of the problem), $N_s$ the number of sample vectors, K the number of steering vectors, and M our sample covariance matrix (not the true one).

We will denote the operations multiplication by *; addition/subtraction by ±; division by /; taking square roots by $\sqrt{}$; moving data by MOVE; and summing data by SUM. $t_{op}(\cdot)$ will denote the time required to perform one scalar (sequential) operation $(\cdot)$. Other operations and times will be introduced as needed.

### 3.2.2 Complex Arithmetic

Since the algorithms involve complex numbers, we must address the problems of how to store them and perform complex multiplication, addition/subtraction, and other operations.

There are two ways of storing complex numbers, Cartesian ($z = x + iy$) and polar ($z = re^{i\theta}$) forms. Polar form is convenient for complex multiplication (requiring one real multiplication and one real addition) but totally inefficient for addition/subtraction, requiring conversion to and from Cartesian form, which requires one square root, one arctangent, one sine, and one cosine. Hence we choose Cartesian form.

Clearly complex additions/subtractions, moving complex data, and multiplying a complex number by either a pure real or pure imaginary number involves two of their real counterparts. Multiplying two complex numbers is more complex and is analyzed in Appendix E, which discusses two methods--one requiring 4 real multiplications and 2 additions/subtractions, and the other requiring 3 real multiplications and 5 real additions/subtractions. We conclude the first method is superior on a sequential machine if and only if

$$t_{op}(*) \leq 3 \cdot t_{op}(\pm) \quad , \tag{3.7}$$

which is satisfied by a majority of machines. Complex division is usually avoidable, and is equivalent to multiplying the numerator by the conjugate of the denominator and dividing by the square of the modulus of the denominator.

### 3.2.3 Inverse Matrix Update (IMU) Algorithm

If $X^{(r)} = \left(X_1^{(r)}, \ldots, X_N^{(r)}\right)^T$ is the $r^{th}$ sample vector, then the maximum likelihood estimator of the true covariance matrix, under the assumption that the underlying distribution function is changing slowly, is [Wilks, 1962]

$$M = \frac{1}{N_{s-1}} \sum_{r=1}^{N_s} \overline{X^{(r)}}\, X^{(r)T} \quad . \tag{3.8}$$

Since there is an arbitrary constant q in Equation 3.3, we may ignore the constant multiplier $1/(N_{s-1})$. Let

$$M_i \triangleq \sum_{r=1}^{i} \overline{X^{(r)}}\, X^{(r)T} \quad . \tag{3.9}$$

Then we have

$$M^{-1}_{i+1} = (M_i + \overline{X}_{i+1} X^T_{i+1})^{-1} = M^{-1}_i - \frac{M^{-1}_i \overline{X}_{i+1} X^T_{i+1} M^{-1}_i}{1 + X^T_i M^{-1}_i \overline{X}_i} \tag{3.10}$$

which is known as Shur's Identity [Shapard, Edelblute, and Kinnison, 1971]. This scheme is usually implemented by letting $M_0 = I$ and then getting

$$M^{-1}_{N_s} = (I + \sum_{r=1}^{N_s} \overline{X^{(r)}}\ X^{(r)T})^{-1} \quad ;$$

and since $N_s$ is usually large ($\approx 2N$), the effects of the starting estimate will be minimal. We may further reduce these effects by introducing a weighting factor $\gamma \approx .99$, yielding

$$M_{i+1}^{-1} = (\gamma M_i + \bar{X}_{i+1} X_{i+1}^T)^{-1} = \frac{1}{\gamma}\left[\frac{M_i^{-1} - (\frac{1}{\gamma})M_i^{-1}\bar{X}_{i+1}X_{i+1}^T M_i^{-1}}{1 + (\frac{1}{\gamma})X_{i+1}^T M_i^{-1}\bar{X}_{i+1}}\right] . \quad 3.11$$

### 3.2.4 Updating the Sample Covariance Matrix

As stated in the last section, under the assumption that the underlying distribution function is changing slowly, the maximum likelihood estimator of the sample covariance matrix is

$$M = \frac{1}{N_s - 1} \sum_{r=1}^{N_s} \overline{X^{(r)}}\, X^{(r)T} \quad . \qquad 3.12$$

This is strictly true only when the underlying distribution is multivariate normal and stationary (constant in time). This estimate degrades as the distribution distorts from normal and changes in time. The greatest danger occurs from the distribution's changing too quickly, preventing M from tracking the time covariance matrix accurately. Let us denote M by $M_{N_s}$ to reflect its dependence on the number of samples.

Equation 3.12 introduces a time bias because it weighs the oldest samples, like $\overline{X^{(1)}}\, X^{(1)T}$, as much as the most recent samples, like $\overline{X^{(N_s)}}\, X^{(N_s)T}$. What is needed is a weighted average of the sample values which satisfies three conditions: 1) it must weigh the recent data values heavily enough so that it follows the current expected value quickly, 2) but it must not weigh them so heavily that noise spikes confuse it, and 3) it must be an unbiased estimator in the best case of a distribution constant with respect to time.

The third condition is easily satisfied by making the sums of the weights equal to $1/\overline{N_{s-1}}$. If we abbreviate $\overline{X^{(r)}}\, X^{(r)T}$ by $\Psi^{(r)}$,

our weighted estimate, $\tilde{M}_{N_S}$, becomes

$$\tilde{M}_{N_S} = \sum_{j=1}^{N_S} C_{N_S j} \Psi^{(j)} \quad . \tag{3.13}$$

Our third condition states that $E(\Psi^{(j)})$ is equal to some constant $\Psi$ for all j such that

$$E(\tilde{M}_{N_S}) = E\left(\sum_{j=1}^{N_S} C_{N_S j} \Psi^{(j)}\right) = \sum_{j=1}^{N_S} C_{N_S j} E\left(\Psi^{(j)}\right)$$

$$= \sum_{j=1}^{N_S} C_{N_S j} \Psi = \Psi \sum_{j=1}^{N_S} C_{N_S j} \quad . \tag{3.14}$$

And, since under the assumption of a constant underlying distribution, $\Psi$ = ($N_S$-1) times the true covariance matrix, we need

$$\sum_{j=1}^{N_S} C_{N_S j} = \frac{1}{N_S - 1} \quad . \tag{3.15}$$

Subject to this mild restriction, we are now free to choose the $C_{N_S j}$ to satisfy the first condition. Making the choice is a much more subtle problem since it depends on the outside environment, and thus could probably require analysis of each system configuration to pick the optimal scheme; but several methods which have worked well in other applications come to mind first.

The simplest method is probably fixed interval reinitialization. This means the input sample stream is divided into groups of T samples

each for some fixed T, and the running average computed at a given time is just the arithmetic mean of available samples in the current group:

$$\tilde{M}_{N_S} = \left( \sum_{j=\left[\frac{N_S-1}{T}\right]T+1}^{N_S} \Psi^{(j)} \right) \Big/ \left(N_S - \left[\frac{N_S-1}{T}\right] T\right) , \qquad 3.16$$

where [X] denotes the greatest integer less than or equal to X. This can be computed recursively by

$$\left.\begin{aligned} &SUM = \Psi^{(N_S)} \\ &NUMBER = 1 \\ &\tilde{M}_{N_S} = SUM \end{aligned}\right\} \quad \text{if } N_S \equiv 1 \bmod T$$

$$\left.\begin{aligned} &SUM = SUM + \Psi^{(N_S)} \\ &NUMBER = NUMBER + 1 \\ &\tilde{M}_{N_S} = SUM/NUMBER \end{aligned}\right\} \text{ otherwise.} \qquad 3.17$$

As mentioned before, we may leave out the constant multiplier without affecting the SNR:

$$\begin{aligned} \tilde{M}_{N_S} &= \Psi^{(N_S)} \quad \text{if } N_S \equiv 1 \bmod T \\ \tilde{M}_{N_S} &= \tilde{M}_{N_{S-1}} + \Psi^{(N_S)} \quad \text{otherwise.} \end{aligned} \qquad 3.18$$

This method has the disadvantage of having very few samples to average over immediately after each reinitialization, which may cause noisy averages.

Slightly more sophisticated is the moving-window scheme, where the current average is the arithmetic mean of the last T samples, for some T:

$$\tilde{M}_{N_S} = \left[ \sum_{j=N_S-T+1}^{N_S} \Psi(j) \right] / T \qquad 3.19$$

(where, as before, the division by T may be omitted). This method is slightly stabler than fixed interval reinitialization but requires a memory large enough to store the last T samples, which are matrices themselves.

By changing the form of our original equation (3.13) to make it recursive, we may write

$$\tilde{M}_{Ns} = C_{N_S N_S} \Psi^{(Ns)} + \sum_{j=1}^{N_S-1} C'_{N_S j} \tilde{M}_j \quad . \qquad 3.20$$

This form has the advantage of requiring a short memory for just a few past values of the running average $\tilde{M}_j$ in order to have $\tilde{M}_{N_S}$ be a function of all past data values and hence be very smooth and stable. The simplest form of this is exponential smoothing:

$$\tilde{M}_{N_S} = \omega\Psi^{(N_S)} + (1 - \omega)\, \tilde{M}_{N_{S-1}} , \qquad 3.21$$

where $0<\omega<1$. If $\omega$ is near 1, the current sample contributes heavily to the current average, so the current average is very sensitive to changes in the environment. If $\omega$ is near 0, the old average is the major

contributor to the current average and so the current average is a very stable estimate insensitive to noise spikes. The optimal choice of $\omega$ depends on the particular system. OTH radar might use a small $\omega$, whereas a close-range radar might work better with a large $\omega$. This method has the advantage of having to save just one old value, $\tilde{M}_{N_{S-1}}$. In terms of computation time, however, it is more expensive than either fixed interval reinitialization or window-smoothing, since it requires two extra multiplications by constants. In practice all such multiplications and divisions in these other two methods can be avoided since the SNR is not affected by a constant multiple of the weights, $\omega$. Since fixed interval reinitialization is the easiest to implement and has provided good results in simulations, we will use it for the balance of the report.

As a last comment, we note that, for $M_{N_S}$ to be nonsingular, it must be a sum of at least N samples, as is proven in Appendix H.

Different methods of forming the outer product $\overline{X}^{(i)}X^{(j)T}$ are discussed in the part of this report on implementations (Part 4).

### 3.2.5 Direct Methods of Solving $MW = \bar{S}$

#### 3.2.5.1 Direct Methods Without Partitioning

Gaussian elimination reduces M to upper triangular form by elementary row operations, and then solves the triangular system by back substitution. Ordinarily, one interchanges rows and columns in order to ensure that the pivots have relatively large absolute values. However, because our matrix is positive definite, no interchanges are necessary [Westlake, 1968, p 13]. If we let $m_{ij}^{(k)}$ be the $(i,j)^{th}$ element of M after the $k^{th}$ pivot, the $k^{th}$ pivot yields:

$$m_{ij}^{(k)} = m_{ij}^{(k-1)} = 0 \qquad i=2, \ldots, N;\ j=1, \ldots, k-1$$

$$m_{ik}^{(k)} = 0 \qquad i=k+1, \ldots, N$$

$$m_{ij}^{(k)} = m_{ij}^{(k-1)} \qquad i=2, \ldots, k;\ j=i, \ldots, N$$

$$m_{ij}^{(k)} = m_{ij}^{(k-1)} - \frac{m_{kj}^{(k-1)}}{m_{kk}^{(k-1)}} \qquad i=k+1, \ldots, N;\ j=k+1, \ldots, N\ .$$

This provides the triangular factorization

$$M = LDL^* \quad ,$$

where L is lower triangular with unit diagonal and D is a diagonal matrix with positive real entries.

Thus, the triangular form is obtained after the $(N-1)^{st}$ pivot. Then, back substitution yields

$$w_i = \frac{1}{m_{ii}^{(N-1)}} \cdot \left[ \bar{s}_i - \sum_{j=i+1}^{N} m_{ij}^{(N-1)} \bar{s}_j \right].$$

A variant of Gaussian Elimination that avoids the back substitution is Gauss-Jordan Elimination. This method eliminates the elements above the diagonal as well as those below. Thus, the first pivot is the same as in Gaussian Elimination, the second pivot also eliminates the (1,2) element, and, in general, the $k^{th}$ pivot eliminates all non-diagonal elements in the $k^{th}$ column. We then find W from the formula

$$w_i = \frac{\bar{s}_i}{m_{ii}^{(N-1)}}.$$

Note that both forms of elimination would work well on any positive definite matrix, and not only on Hermitian ones. A direct method that requires this sesquisymmetry is Cholesky's Factorization Method which, because it takes advantage of the Hermitian properties, requires the fewest (serial) operations of any direct method without partitioning.

Cholesky's method is a straightforward application of the LDU theorem for Hermitian matrices, which states: If M is Hermitian and arranged so that none of its upper-left principal submatrices is singular, then $M = LL^*$, where L is a unique lower triangular matrix, and $L^*$ is its conjugate transpose [Westlake, 1968, p 126]. Since every upper-left principal submatrix (i.e., the first j rows and j columns) of a positive

definite matrix is itself positive definite, and thereby non-singular, we see that Cholesky's Method takes advantage of all properties that M, a priori, possesses. The elements of L are given by the following formulas [Wilkinson and Reinsch, 1971, p 9]:

$$\ell_{ii} = \left( m_{ii} - \sum_{k=1}^{i-1} \ell_{ik} \cdot \overline{\ell}_{ik} \right)^{1/2} \quad i=1, \ldots, N \qquad 3.22$$

$$\ell_{ij} = \frac{1}{\ell_{jj}} \left( m_{ij} - \sum_{k=1}^{j-1} \ell_{ik} \cdot \overline{\ell}_{jk} \right) \quad i=1, \ldots, N, \; j=1, \ldots, i-1 \; .$$

Thus, we transform Eq. (3.3) into

$$LL^*W = \overline{S} \; . \qquad 3.23$$

By slightly changing Equation 3.22, we may instead perform the decomposition

$$LDL^*W = \overline{S} \quad , \qquad 3.24$$

where L has a unit diagonal and D is a diagonal matrix with positive real entries:

$$d_i = m_{ii} - \sum_{k=1}^{i-1} \tilde{m}_{ik} \ell_{ik} \qquad 3.25$$

$$\tilde{m}_{ij} = m_{ij} - \sum_{k=1}^{j-1} \tilde{m}_{ik} \ell_{jk} \quad j=1, \ldots, i-1$$

$$\ell_{ij} = \tilde{m}_{ij}/d_j \quad .$$

These methods may be used to directly solve for the weights by two back substitutions:

1) solve $LT = \overline{S}$ and $L^*W = T$ when $M = LL^*$

2) solve $LT = \overline{S}$, $DQ = T$, and $L^*W = Q$ when $M = LDL^*$ .

If, in place of $\overline{S}$, one uses the identity matrix I, the first back substitution $LT = I$ may be considerably shortened by taking advantage of the special form of I. In this way, by performing the other back substitution, $M^{-1}$ may be computed, but it still must be multiplied by $\overline{S}$. Thus GE and both versions of Cholesky can be used to solve $MW = \overline{S}$ in two ways. GJ may also be used to form $M^{-1}$ by substituting I for $\overline{S}$. It may be shown, however, that, on a sequential machine, forming $M^{-1}$ is inferior to two back substitutions [Isaacson and Keller, 1966].

### 3.2.5.2 Direct Methods With Partitioning

Partitioning can be used either to improve the asymptotic computational complexity [Strassen, 1969] or to reduce the problem's dimension to a size especially convenient for the processor [Troyer, 1968]. Its disadvantages include increased overhead to keep track of partitions and intermediate results, and, depending on the problem size and implementation, large stacks of intermediate results which require a great deal of core.

Bunch and Hopcroft [1974] have shown matrix multiplication, triangular factorization, and inversion are equivalent in computational complexity; i.e., if an $O(N^{\alpha})$ algorithm, where $2<\alpha\leq 3$, can be found for one of these tasks, $O(N^{\alpha})$ algorithms can be found for the other two. They also improved the constant factor in Strassen's original algorithm. For the best case when N is a power of 2, the dominating term in the total operation count (multiplication, subtraction, addition and division together) for their algorithm for LU factorization is $2.04\ N^{\log_2 7}$. The dominating term of the total operation count for Cholesky decomposition is $\frac{1}{3} N^3$. For the partition method to be superior to Cholesky's, we must have

$$2.04\ N^{\log_2 7} < \frac{1}{3} N^3$$

or

$$N > 12{,}132 \quad ,$$

which is far beyond the range of N we are considering.

### 3.2.6 Iterative Methods of Solving $M\bar{W} = \bar{S}$

The simplest of these methods is the Point-Jacobi method. We write M as D-U-L, where D is diagonal, U is strictly upper triangular, and L is strictly lower triangular (note that $L^T=\bar{U}$ ). We then have

$$(D-U-L)W=\bar{S} \quad .$$

or

$$DW=(U+L)W+\bar{S} \quad .$$

Since D has all non-zero entries, we get

$$W=D^{-1}(U+L)W+D^{-1}\bar{S} \quad .$$

This yields the basic iterative scheme, which is

$$W^{(k+1)} = D^{-1}(U+L)W^{(k)} + D^{-1}\bar{S} \qquad 3.26$$

or

$$w_i^{(k+1)} = \frac{-1}{m_{ii}} \left[ \sum_{\substack{j=1 \\ j\neq i}}^{N} m_{ij} w_j^{(k)} \right] + \frac{\bar{s}_i}{m_{ii}} \quad . \qquad 3.27$$

The second stationary technique, the Gauss-Seidel method, is a simple variant of the Point-Jacobi method. From Eq. 3.27, one sees that the $(k+1)^{st}$ estimates for $w_j^{(k+1)}$, j=1, ..., i-1 are available for use in the computation of $w_i^{(k+1)}$, and the Gauss-Seidel method merely uses this fact.

We then get

$$w_i^{(k+1)} = \frac{-1}{m_{ii}} \left[ \sum_{j=1}^{i-1} m_{ij} w_j^{(k+1)} \right] - \frac{1}{m_{ii}} \left[ \sum_{j=i+1}^{N} m_{ij} w_j^{(k)} \right] + \frac{\bar{s}_i}{m_{ii}} , \tag{3.28}$$

which in matrix notation becomes

$$W^{(k+1)} = (D-L)^{-1} U W^{(k)} + (D-L)^{-1} \bar{S} . \tag{3.29}$$

Were we to try to implement the matrix form, however, we would have the additional difficulty of inverting a triangular matrix.

If we add an additional refinement, we get the third stationary method. We can let $W^{(m+1)}$ be a weighted average of $\hat{W}^{(m+1)}$ and $W^{(m)}$, where $\hat{W}^{(m+1)}$ is the result of a Gauss-Seidel iteration on $W^{(m)}$. Thus, $W^{(1)}$ in this method is merely a weighted average of $W^{(1)}$ from the Gauss-Seidel method, and $W^{(0)}$, the initial "guess." This method, the point successive overrelaxation iterative method (abbreviated SOR), is characterized by the following two equations:

$$W^{(m+1)} = (I-\gamma D^{-1} L)^{-1} [(1-\gamma) I + \gamma D^{-1} U] W^{(m)} + \gamma [I - D^{-1} L] D^{-1} \bar{S} \tag{3.30}$$

$$w_i^{(m+1)} = w_i^{(m)} - \frac{\gamma}{m_{ii}} \left[ \sum_{j=1}^{j-1} m_{ij} w_j^{(m+1)} + \sum_{j=i+1}^{N} m_{ij} w_j^{(m)} + \bar{s}_i - m_{ii} x_i^m \right] . \tag{3.31}$$

Setting $\gamma=1$ degenerates to the Gauss-Seidel method.

We have the following theorems about convergence:

T1: The Point-Jacobi and Gauss-Seidel methods either both converge or both diverge [Varga, 1962, p 70].

T2: The SOR method converges for all $\gamma$, $0 < \gamma < 2$, or it converges for none [Varga, 1962, p 80].

T3: A necessary and sufficient condition for the Gauss-Seidel method to converge is the positive definiteness of M [Varga, 1962, p 78].

These three theorems together ensure that, because M is positive definite, the Point-Jacobi and Gauss-Seidel methods will both converge, as will the SOR method if $0 < \gamma < 2$.

We can estimate relative rates of convergence only roughly. For example, we know that the Gauss-Seidel method asymptotically converges twice as fast as the Point-Jacobi method. However, we have no a priori knowledge concerning the relative degree of convergence after a fixed number of iterations. The SOR method converges significantly faster than both other stationary methods, if a good choice of $\gamma$ is made. In general, $\gamma$ must be found experimentally, although we know that $1 < \gamma < 2$ [Westlake, 1968, p 63].

Several authors [Varga, Westlake, Young] suggest using iterative methods only when rapid convergence is ensured, for example, with large sparse matrices. One study by Young suggests that the SOR method requires on the order of 3N iterations for reasonable convergence [Young, 1973, p 117].

The nonstationary, linear technique we shall discuss is called Chaotic Relaxation, and was developed by Chazan and Miranker [1969]. It was designed to be implemented on a multiprocessor system with common memory. Essentially, each processor updates one component of the W vector, and once done returns to memory and finds another component to update. We specifically do <u>not</u> assume that the components are updated in serial order, or even in <u>any</u> predetermined order, hence the name Chaotic Relaxation. The general scheme can be characterized by:

$$w_i^{(k+1)} = \sum_{j=1}^{N} p_{ij} w_j^{[k-\ell(k,j)]} + q_i \qquad i=h(k)$$

$$w_i^{(k+1)} = w_i^{(k)} \qquad i\neq h(k) \quad ,$$

where $h(k)$ is the component altered in the $k^{th}$ updating, and the value of $w_j$ used to calculate it is taken from the $k-\ell(k,j)$ iteration. We assume that the unknown function $h(k)$ takes each value between 1 and N infinitely often, and also that $\ell(k,j) < c$ for some fixed c and all k and j. Chazan and Miranker have shown (in the real case, although the proof in the complex case is analogous) that if P is the iteration matrix from the Point-Jacobi, Gauss-Seidel, or SOR methods, then the Chaotic scheme will converge. They do not, however, present any comparisons of convergence rates. The only substantial benefit which Chaotic Relaxation offers is the elimination of the computer's "bookkeeping" tasks. In high-dimension systems, however, these "bookkeeping" tasks require proportionately less and less of the total time,

and so it is unlikely that the Chaotic Relaxation method will offer much improvement over the standard, stationary techniques [see also Donnelly, 1971].

### 3.2.7 Parallel Methods

There are two general classes of parallel methods: those which were initially sequential algorithms (all those methods discussed so far) but were adapted to exploit their inherent parallelism, and those which were designed specifically for parallel implementation. Much material has been published on parallel methods [Sameh and Kuck, 1975; Berra, 1976; Pace, 1972] but almost all of it has discussed the first class of methods only: parallel versions of Gauss-Jordan, Gaussian Elimination, etc. We found only one specifically parallel algorithm, by Csanky [1974 and 1976].

Csanky's algorithm is worthy of note because it has the lowest computational complexity (number of steps, where each step may include several computations performed simultaneously) of any algorithm published to date: $O(\log^2 N)$. It had a major drawback that eliminated it from further consideration: it requires $O(N^5)$ parallel processors. For a system of N = 200 weights, approximately $10^{11}$ processors would be required.

Because of this algorithm's low computational complexity, we decided to explore the effect of the number of parallel processors on it, in the hope that its complexity would not degrade very much by the time a practical number of processors was reached.

As will be shown in the section on parallel implementations, Gauss-Jordan can be performed in O(N) steps using $N^2$ parallel processors. We will compare this result with Csanky's algorithm. Assume that only $N^4$ processors are available for performing Csanky's algorithm. Since at least one step must require all $O(N^5)$ processors, that step will expand to $O(N^5)/N^4 = O(N)$ steps. Hence Gauss-Jordan can achieve approximately the

same computational complexity, $O(N)$, as Csanky's algorithm, but with only $O(N^2)$ processors instead of $O(N^4)$. Hence Csanky's algorithm degrades too rapidly as a function of the number of processors to be of interest for large N, but may deserve attention for small N, where it is practical to produce all $O(N^5)$ processors.

We can generalize the above analysis. Assume an algorithm requires $p(N)$ parallel processors to achieve a computational complexity of $O(c(N))$. If only $p_1(N) < p(N)$ processors are available, then, since at least one step requires all $p(N)$ processors, it will expand to $p(N)/p_1(N)$ steps, so the computational complexity will be at least $O(p(N)/p_1(N))$ and could be as high as $O(c(N)p(N)/p_1(N))$ if all the steps require all processors. So, for example, if $p(N) = N^m$ and $p_1(N) = N^{m-1}$, the computational complexity will be at least $O(p(N)/p_1(N)) = O(N)$, which we know we can achieve with Gauss-Jordan and $O(N^2)$ processors, as mentioned above.

It has been shown [Borodin and Munro, 1975, Theorem 6.1.3; Csanky, 1976, Lemma 1] that at least $2 \cdot \log N$ steps are required to either invert an order N matrix, solve a system of equations of order N, compute an order N determinant, or determine the characteristic equation of a matrix of order N. It has not been shown whether this lower bound can be achieved, and a great deal of work remains to be done to fill the gap between Csanky's $O(\log^2 N)$ and the optimal $O(\log N)$.

### 3.2.8 The Gram-Schmidt Technique

As shown in Section 3.1, the filter function we want to calculate is

$$F = W^T X = (M^{-1}\bar{S})^T X = S^*(M^{-1})^T X = S^*\overline{M^{-1}}\, X \qquad 3.32$$

Since it is difficult to compute $M^{-1}$ (or solve $MW = \bar{S}$ for W), it would be advantageous to transform Equation 3.32 into a more quickly computable form. Let us denote the sample covariance matrix of the X's by $M_X$ instead of M, and let T be any nonsingular linear transformation. Define $\mathcal{S}$ and Y by $\mathcal{S} = TS$ and $Y = TX$. Substituting these in Equation 3.32 we obtain

$$F = S^*\,\overline{M_X^{-1}}\, X = (T^{-1}\mathcal{S})^*\overline{M_X^{-1}}\,(T^{-1}Y) = \mathcal{S}^*\overline{(\bar{T}\,M_X\,T^T)^{-1}}\, Y \quad . \qquad 3.33$$

It can easily be shown that if $M_X$ is the covariance matrix of X, $M_Y$ is the covariance matrix of Y, and $Y = TX$, then

$$M_Y = \bar{T}\,M_X T^T$$

Substituting this result in Equation 3.33, we have

$$F = \mathcal{S}^*\,\overline{M_Y^{-1}}\, Y \quad .$$

Note that this equation is analogous to our original untransformed equation, 3.32.

Our only restriction on T is that it be nonsingular. If we could choose T so that $M_Y$ were easily invertible (in particular, diagonal) and so that TX were easily computable, we would have greatly simplified our problem. This approach leads to the concept of orthogonality.

In our case, the orthogonality of two components of our sample vector is equivalent to their being uncorrelated:

$$(X_j, X_i) = E(\bar{X}_i X_j) = 0.$$

Hence, we can reduce the problem to choosing a transformation, T, that satisfies the following conditions:

1) T is nonsingular .

2) The vector Y = TX is easy to compute and has sufficiently uncorrelated components.

3) The components of T are easy to form and update with new data.

The Gram-Schmidt Orthogonalization Process, which is a method of transforming a set of components (or vectors of random variables) into an equivalent set of uncorrelated (orthogonal) components, shows great promise as an algorithm to choose T. Gram-Schmidt will not be discussed further in this report.

### 3.2.9 Operation Counts and Conclusions

The direct methods without partitioning, and of these Cholesky in particular, seem to be best for a sequential or almost-sequential machine. Table 3.1 contains complex operation counts for these methods (Cholesky may or may not use N square roots). In addition, Cholesky is an extremely stable process numerically [Wilkinson and Reinsch, 1971].

Table 3.1 Complex Operation Counts of Direct Methods Without Partitioning [Westlake, 1968, p 100]

| | $\star$ | $\pm$ | $\frac{1}{}$ | $\sqrt{\ }$ |
|---|---|---|---|---|
| GE | $\frac{1}{3}N^3 + N^2 - \frac{1}{3}N$ | $\frac{1}{3}N^3 + N^2 - \frac{5}{6}N$ | N | 0 |
| GJ | $\frac{1}{2}N^3 + N^2 - \frac{1}{2}N$ | $\frac{1}{2}N^3 - \frac{1}{2}N$ | N | 0 |
| Cholesky | $\frac{1}{6}N^2 + \frac{3}{2}N^2 + \frac{1}{3}N$ | $\frac{1}{6}N^3 + N^2 - \frac{7}{6}N$ | N | (N) |

Following decomposition by Cholesky (or any other method), two back substitutions are recommended as superior to forming the inverse and multiplying by the steering vector [Isaacson and Keller, 1966].

We do not believe any of the iterative schemes are worth pursuing for the following three reasons:

1) The number of iterations required for suitable convergence is highly dependent on the particular matrix components, and thus may vary greatly from matrix to matrix. Consequently, no total operations count, nor time estimates, may be given. Rather, we could only offer an operations-per-iteration count, or time-per-iteration estimate.

2) In order to be reasonably efficient, iterative schemes require a weighting factor $\gamma$. In order to calculate $\gamma$, a good estimate of the eigenvalues of the iteration matrix is necessary, and that is something we do not have [Westlake, Varga].

3) Each iteration requires on the order of $N^2$ operations. Unless one has reason to believe that convergence will be rapid (say, because the matrix is sparse), reasonable convergence frequently takes more than N iterations, thus raising the total number of operations above $N^3$ [Young, 1973, p 147].

We also recommend against IMU (on a sequential machine, at least) because its highest-order term is $N_s \cdot 2N^2$ multiplications (to process $N_s$ samples) and $N_s$ is usually about 2N so the dominating term is $4N^3$. Using fixed interval reinitialization to compute the covariance matrix, Cholesky requires $N_s N^2 + \frac{1}{6} N^3$ multiplications, or $\frac{13}{6}N^3$ when $N_s=2N$, and is thus far superior to IMU. In fact for IMU to require fewer multiplications than Cholesky we would need $2N^2N_s < \frac{1}{6}N^3 + N^2N_s$ or $N_s < \frac{1}{6}N$, which is far too few samples to even have a nonsingular sample covariance matrix.

The partitioning methods, while asymptotically superior, also are inferior to Cholesky for our range of N.

The parallel algorithms are potentially much superior to any sequential method, even if advantage is taken of the inherent parallelism in the sequential methods. However, on a sequential

machine, the parallel methods are generally far inferior to any sequential algorithm. Thus, algorithm choice becomes processor-specific and the subject of the next part of this report.

# 4.0 IMPLEMENTATIONS

## 4.1 INTRODUCTION

In this part of the report we will analyze the algorithms of the previous sections as they are implemented on computers with different architectures. The simple operation counts of the last part will no longer be valid criteria by which the best implementation may be selected. The same algorithm may have different implementations in different architectures as well as different implementations in the same architecture, depending on the storage scheme for the matrix, for example. We have attempted to broaden the operation counts approach by including operations specific a certain architecture, such as startups for a vector pipeline processor and processor enables for parallel machines. To get an accurate timing estimate, however, it is necessary to code the program and run it. The compiler or assembler used produces overhead which is impossible to predict accurately and which contributes to the execution time. We chose a high-level language, FORTRAN, to code our implementations, for several reasons. First, it provides a reasonable upper bound on the speed of the implementation since the extra overhead contributed by the high-level language in contrast to assembly or machine language is small compared to the time spent in performing the computations. The times are also still valid for comparing different implementations. Second, FORTRAN is much simpler and cheaper to use than assembly language, and it is much easier to maintain and prove correctness of programs written in a high-level language. This is the attitude of DoD directives 5000.29 and 5000.30.

The rest of this part is organized in sections devoted to particular architectures. Each section includes an introduction to the architecture and some of its hardware aspects; the different algorithms and their corresponding implementations, along with operation counts and conclusions; and finally, the results of test runs on actual machines, including both simulations to determine the SNR attainable and timing runs.

## 4.2 SEQUENTIAL PROCESSORS

### 4.2.1 Introduction to Sequential Processors

A sequential processor is what one first thinks of when one thinks of a computer: a machine for which the time required to perform a sequence of operations, T, is simply the number of operations, n, times the time required to perform one operation, $t_{sop}(X)$

$$T = n \cdot t_{sop}(X) \quad . \qquad 4.1$$

The subscript "sop" denotes "scalar operation" and "X" is the actual operation. This equation implies that operation counts are completely sufficient to rank algorithms by the time they require.

There are few large sequential processors, however, which satisfy Equation 4.1 in all cases. There are factors--such as memory conflicts, instruction stack size, the relative times it takes to branch to an instruction in the stack versus an instruction outside the stack, the existence of a fast, small core memory (SCM) and a slow, large core memory (LCM), or a mass storage device (disk, drum) among which data and instructions are swapped, the number and type of registers, the number, type, and speed of I/O channels, whether or not memory is interleaved, and the existence of cache memory--which affect the speed of all computers, not just sequential ones. All these factors affect the rates at which operands can be fed to the part of the CPU which performs the (arithmetic) operations on them, and the rate at which the results can be restored in memory. These rates are determined, in part, by the number of instructions and the order in which they are executed, and the amount of data, its storage scheme, and access pattern. Both rates may be input-dependent, if any branching is done based on the input data, for example.

There is another factor, however, which is typical of sophisticated sequential processors, which makes the naive use of operation counts to compare algorithms unsafe. This factor is the degree of parallelism available in the CPU, particularly, whether or not the CPU can overlap the execution of certain instructions.

For example, if the CPU can perform a multiplication and addition simultaneously, it is to an algorithm's advantage to mingle its operations. So, assuming a multiplication and addition each take 1 unit of time, Algorithm A in Fig. 4.1 requires only 3 units of time compared to Algorithm B which takes 4 units. In Algorithm A, steps 1) and 2) are performed simultaneously, followed by 3) and 4) simultaneously, and finally 5). In Algorithm B the only instruction overlap is that of steps 2) and 3). Similarly, if the machine can overlap on addition and a jump (GOTO) then Algorithm C (in Fig. 4.2) is superior to Algorithm D.

Another feature is look-ahead processing. This feature helps in case instruction overlap is possible and there is a string of operations each of which uses the result of the previous operation as an input. The look-ahead feature scans ahead in the instruction stream, searching for an instruction which can be executed independent of the current instruction, and then executes it. This feature is discussed by Keller [1975].

### 4.2.2 Implementations of Algorithms to Determine Weights on a Sequential Processor

The basic methods for determining adaptive weights are the inverse matrix update method (IMU), forming the sample covariance matrix M and solving $MW = \overline{S}$, and loops. In the second method $MW = \overline{S}$ may be solved by either direct or iterative methods. We will not discuss loops or solving $MW = \overline{S}$ by iterative methods because of their slow or indeterminate convergence rates, as discussed in previous sections.

| Algorithm A | Algorithm B |
|---|---|
| 1) A = B*C | 1) A = B*C |
| 2) D = A+E | 2) F = G*H |
| 3) F = G*H | 3) D = A+E |
| 4) S = F+R | 4) S = F+R |
| 5) T = S+D | 5) T = S+D |

Figure 4.1 Different Algorithms for Computing T = B*C + E + G*H + R on a Sequential Processor with Instruction Overlap

| Algorithm C | Algorithm D |
|---|---|
| S=0<br>J=1<br>10 IF(J.GT.N)GOTO 20<br>S=S+A(J)<br>J=J+1<br>GOTO 10<br>20 CONTINUE | S=0<br>J=0<br>10 J=J+1<br>S=S+A(J)<br>IF(J.LT.N)GOTO 10 |

Figure 4.2 Different Algorithms for Computing $S = \sum_{J=1}^{N} A(J)$ on a Sequential Processor with Instruction Overlap

Our means of comparison will be operation counts. The considerations of Section 4.2.1 are computer- and system-dependent and hence further discussion of them belongs in the next section, on implementations for specific computers. The operations we will count include multiplication (*), addition/subtraction ($\pm$), reciprocation (/), taking square roots ($\sqrt{\ }$), and moving data (MOVE). Only floating-point operations are counted; overhead (index calculations) is computer-dependent. The operations may have real or complex operands, but for the sake of operation counting, all have been broken down into real operations under the assumptions of the next paragraph.

Since algorithms have different implementations and hence running times, depending on the computer and system configuration, we must make several assumptions here to preserve the generality of the discussion. We have tried to make the fewest assumptions possible about special functions or features available, or the amount, format, and timing of data availability. We must also consider complex multiplication, since, as is shown in Appendix E, there are two basic methods, one requiring four real multiplications and two real additions (and subtractions), and one requiring three multiplications and five additions/subtractions. The tradeoff point or the condition to be satisfied for the first algorithm to be superior is:

$$t_{sop}(*)/t_{sop}(\pm) \leq 3 \quad ,$$

where $t_{sop}(\cdot)$ is the time required to perform one operation $(\cdot)$. This condition is satisfied by most current sequential processors (the CDC 7600, for example) and hence in the real operation counts we assume a complex multiplication is transformed into 4 real multiplications and 2 real

additions/subtractions. Multiplying a complex number by a real or pure imaginary number is equivalent to two real multiplications, of course. We have tried to take advantage of situations where we know a priori that the imaginary (or real) part of a product will be zero, such as when multiplying a complex number by its complex conjugate. Complex additions and moves, of course, are equivalent to two of their real counterparts.

There are very few assumptions or restrictions with respect to the system configuration. Since the covariance matrix is Hermitian, it is only necessary to compute and store half of it. When the problem dimension (number of weights) is large, this storage scheme may be necessary to fit the entire matrix in core. Some algorithms do not permit this storage scheme; the ones we will eventually select do.

Recalling the operation counts from the section on mathematical techniques, we conclude that IMU, and solving $MW = \bar{S}$ by Gaussian elimination, Gauss-Jordan elimination or one of the $O(n^{\log 2^7})$ methods developed initially by Strassen are all inferior to Cholesky factorization followed by two back substitutions (or augmented decomposition followed by one back substitution).

The operation counts for the two versions of Cholesky, LDL* and LL*, are shown in Table4.1, and the corresponding implementations are given in Appendix F, along with the counts and implementations for computing the sample covariance matrix and the two back substitutions. The real and imaginary parts of all matrices are stored separately, the covariance matrix rowwise, and the steering vectors vectorwise, as shown in Fig. 4.3.

The only real tradeoffs are between LDL* Decomposition and LL* Decomposition, the former requiring approximately $N^2$ move operations while the latter requires N square roots. The timing of each individual processor must be consulted to choose the best algorithm.

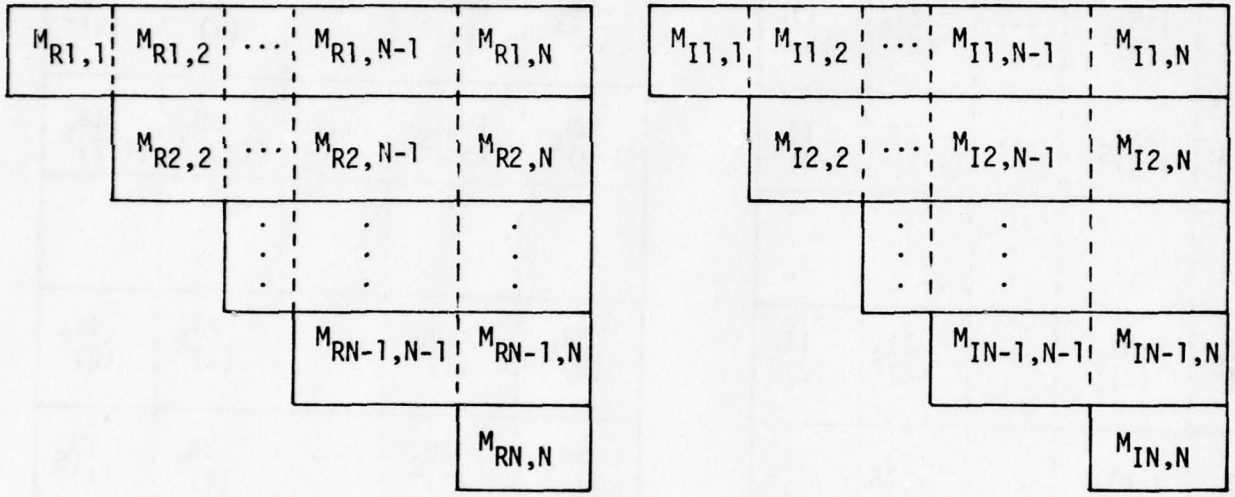
Table 4.1 Real Operation Counts for Sequential Processor Implementations for Determining Adaptive Weights

(N = number of weights, $N_S$ = number of sample vectors, K = number of steering vectors)

| | * | $\overset{+}{-}$ | MOVE | / | $\sqrt{\ }$ |
|---|---|---|---|---|---|
| Update Sample Covariance Matrix | $2N^2N_S$ | $2N^2N_S$ | 0 | 0 | 0 |
| Cholesky (LDL*) Decomposition | $\frac{2}{3}N^3 - \frac{8}{3}N+4$ | $\frac{2}{3}N^3-N^2-\frac{5}{3}N+4$ | $N^2-N$ | N | 0 |
| Cholesky (LL*) Decomposition | $\frac{2}{3}N^3 - \frac{8}{3}N+4$ | $\frac{2}{3}N^3-N^2-\frac{5}{3}N+4$ | 0 | N | N |
| First Back Substitution (LDL*) | $2KN^2$ | $2KN^2 - 2KN$ | 0 | 0 | 0 |
| First Back Substitution (LL*) | $2KN^2$ | $2KN^2 - 2KN$ | 0 | 0 | 0 |
| Second Back Substitution (LDL*) | $2KN^2 - 2KN$ | $2KN^2 - 2KN$ | 0 | 0 | 0 |
| Second Back Substitution (LL*) | $2KN^2$ | $2KN^2 - 2KN$ | 0 | 0 | 0 |

REAL PART IMAGINARY PART

Figure 4.3a Sequential Processor Storage Scheme for the Sample Voltage Vector X: $X_j = X_{Rj} + iX_{Ij}$, $1 \leq j \leq N$




REAL PART IMAGINARY PART

Figure 4.3b Sequential Processor Storage Scheme for the Sample Covariance Matrix M: $M_{j,K} = M_{Rj,K} + i\, M_{Ij,K}$, $1 \leq j, k \leq N$

| | | | | |
|---|---|---|---|---|
| $S_{R1}^{(1)}$ | $S_{R2}^{(1)}$ | ... | $S_{RN-1}^{(1)}$ | $S_{RN}^{(1)}$ |
| $S_{R1}^{(2)}$ | $S_{R2}^{(2)}$ | ... | $S_{RN-1}^{(2)}$ | $S_{RN}^{(2)}$ |
| . | . | . | . | . |
| . | . | . | . | . |
| . | . | . | . | . |
| $S_{R1}^{(K-1)}$ | $S_{R2}^{(K-1)}$ | ... | $S_{RN-1}^{(K-1)}$ | $S_{RN}^{(K-1)}$ |
| $S_{R1}^{(K)}$ | $S_{R2}^{(K)}$ | ... | $S_{RN-1}^{(K)}$ | $S_{RN}^{(K)}$ |

REAL PART

| | | | | |
|---|---|---|---|---|
| $S_{I1}^{(1)}$ | $S_{I2}^{(1)}$ | ... | $S_{IN-1}^{(1)}$ | $S_{IN}^{(1)}$ |
| $S_{I1}^{(2)}$ | $S_{I2}^{(2)}$ | ... | $S_{IN-1}^{(2)}$ | $S_{IN}^{(2)}$ |
| . | . | . | . | . |
| . | . | . | . | . |
| . | . | . | . | . |
| $S_{I1}^{(K-1)}$ | $S_{I2}^{(K-1)}$ | ... | $S_{IN-1}^{(K-1)}$ | $S_{IN}^{(K-1)}$ |
| $S_{I1}^{(K)}$ | $S_{I2}^{(K)}$ | ... | $S_{IN-1}^{(K)}$ | $S_{2N}^{(K)}$ |

IMAGINARY PART

Figure 4.3c Sequential Processor Storage Scheme for the Steering Vectors $S^{(\ell)}$

$S_j^{(\ell)} = S_{Rj}^{(\ell)} + i\, S_{Ij}^{(\ell)}$, $1 \leq j \leq N$, $1 \leq \ell \leq K$

### 4.2.3 An Implementation of an Algorithm for Determining Adaptive Weights on a Sequential Processor--the CDC 7600

The following summary of the main features of the CDC 7600 was extracted from the Control Data 7600 Series CYBER 70/Model 76 Computer Systems: Hardware Manual [1975].

The CDC 7600 consists of a central processor and peripheral processor units (PPUs). The central processor consists of a central processing unit (CPU), functional units, central or small core memory (SCM), large core memory (LCM), and an input/output multiplexer (MUX). See Figure 4.4.

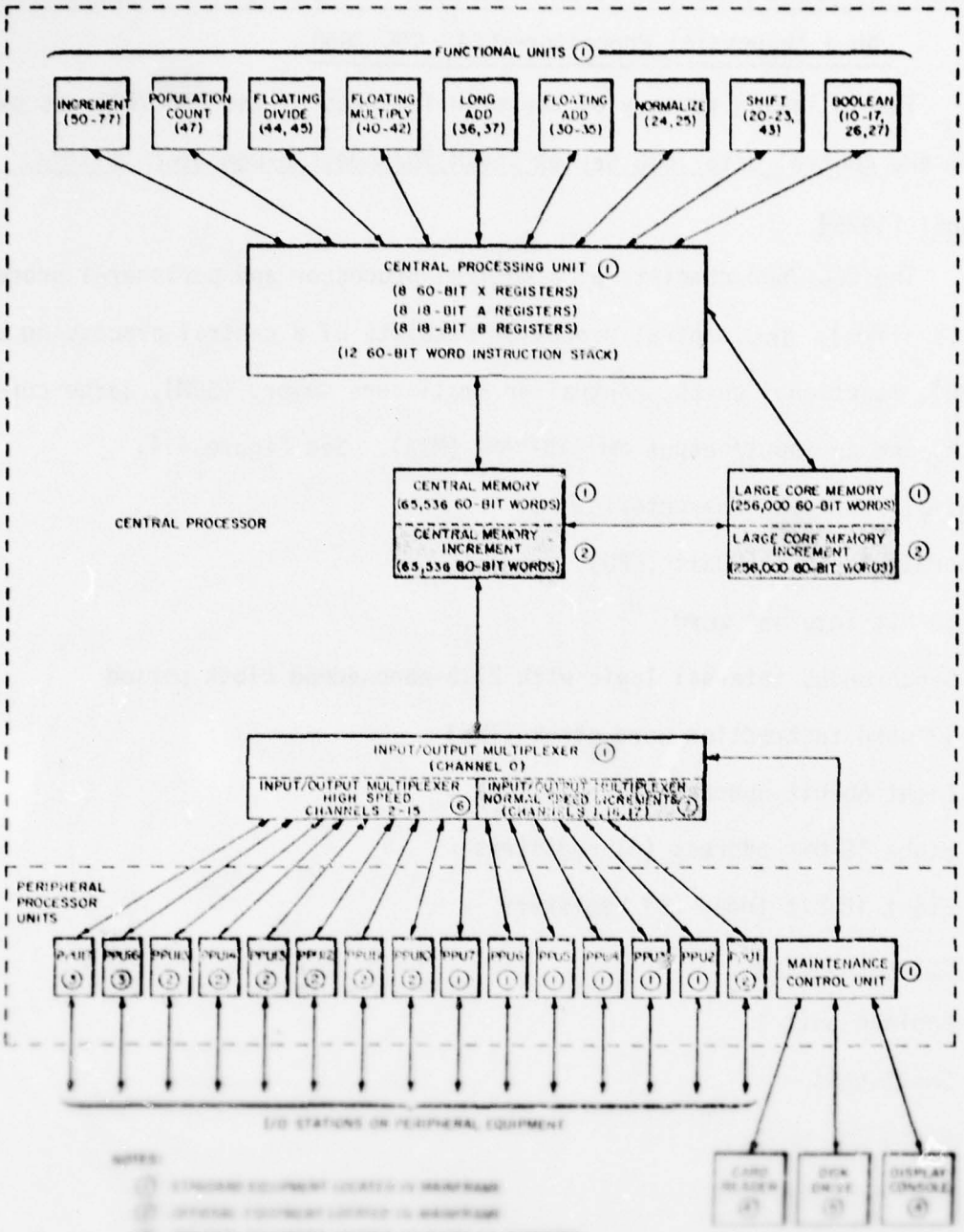
Central Processor Characteristics

Central Processing Unit (CPU)

- 60-bit internal word
- Synchronous internal logic with 27.5-nanosecond clock period
- 12-word instruction word stack (IWS)
- Eight 60-bit operand (X) registers
- Eight 18-bit address (A) registers
- Eight 18-bit index (B) registers

Functional Units

- Boolean unit
- Shift unit
- Normalize unit
- Floating add unit
- Long add unit
- Floating multiply unit
- Floating divide unit
- Population count unit
- Increment unit


FUNCTIONAL UNITS (1)
INCREMENT (50-77)
POPULATION COUNT (47)
FLOATING DIVIDE (44,45)
FLOATING MULTIPLY (40-42)
LONG ADD (36,37)
FLOATING ADD (30-35)
NORMALIZE (24,25)
SHIFT (20-23, 43)
BOOLEAN (10-17, 26,27)
CENTRAL PROCESSING UNIT (1)
(8 60-BIT X REGISTERS)
(8 18-BIT A REGISTERS)
(8 18-BIT B REGISTERS)
(12 60-BIT WORD INSTRUCTION STACK)
CENTRAL PROCESSOR
CENTRAL MEMORY (65,536 60-BIT WORDS) (1)
CENTRAL MEMORY INCREMENT (65,536 60-BIT WORDS) (2)
LARGE CORE MEMORY (256,000 60-BIT WORDS) (1)
LARGE CORE MEMORY INCREMENT (256,000 60-BIT WORDS) (2)
INPUT/OUTPUT MULTIPLEXER (1) (CHANNEL 0)
INPUT/OUTPUT MULTIPLEXER HIGH SPEED CHANNELS 2-15 (6)
INPUT/OUTPUT MULTIPLEXER NORMAL SPEED INCREMENTS (CHANNELS 1, 16, 17) (2)
PERIPHERAL PROCESSOR UNITS
PPU11 (1)
PPU10 (3)
PPU9 (2)
PPU8 (2)
PPU7 (2)
PPU6 (2)
PPU5 (2)
PPU4 (2)
PPU3 (1)
PPU2 (1)
PPU1 (1)
PPU0 (1)
PPU1 (1)
PPU2 (1)
PPU3 (2)
MAINTENANCE CONTROL UNIT (1)
I/O STATIONS OR PERIPHERAL EQUIPMENT

Input/Output Multiplexer (MUX)

The following description applies to the I/O MUX used in systems with semiconductor central memory.

- Sixteen bidirectional channels available

  Channel 0 connects to MCU

  Channels 2 through 15 (octal) are high-speed channels

  Channels 1, 16, and 17 (octal) are normal-speed channels
- Seven high-speed channels and channel 0 (MCU) included in basic system
- Five additional high-speed and three normal-speed channels available as options
- Optional channels may be installed individually or in any combination
- Fixed 128-word buffer areas in semiconductor memory for high-speed channels 10 through 15 and normal-speed channels 1, 16, and 17
- Fixed 256-word buffer areas in semiconductor memory for high-speed channels 2 through 7
- Any area in semiconductor memory addressable by the MCU via channel 0.

Systems with small core central memory have the following I/O MUX configuration.

- 8, 12, or 16 bidirectional 12-bit channels

  Channel 0 connects to MCU

  Channels 1 and 10 through 17 (octal) are normal-speed channels

  Channels 2 through 7 are high-speed channels
- Fixed 128-word buffer areas in small core memory for normal-speed channels 1 and 10 through 17
- Fixed 256-word buffer areas for high-speed channels 2 through 7
- Any area in small core memory addressable by the MCU via channel 0.

Central Memory (SCM)

- Two types: semiconductor memory in most recent models; small core memory in earlier models
- Minimum size: 32,768 60-bit words for small core memory
  65,536 60-bit words for semiconductor memory
- Small core memory organized as 16 or 32 independent 2K banks
- Small core memory expandable to 65,536 60-bit words
- Semiconductor memory organized as 16 or 32 independent 4K banks
- Semiconductor memory expandable to 131,072 60-bit words
- Basic memory cycle time within a bank:
  * Semiconductor memory - 110-nanosecond read, 165-nanosecond write
  * Small core memory - 275-nanosecond read or write
- Phased bank organization yields very fast transfer rate time for references to sequential addresses:
  * 27.5-nanosecond-per-word maximum transfer rate for both semiconductor memory and small core memory
- Semiconductor memory includes single error correction/double error detection (SEC/DED) feature

Large Core Memory (LCM)

- 256,000 or 512,000 60-bit words of linear select memory with four parity bits per 60-bit word
- Organized into four or eight independent banks (64,000 words per bank)
- 1760-nanosecond read/write cycle time
- Eight words read simultaneously each reference
- 27.5-nanosecond-per-word maximum transfer rate (with 512,000-word LCM)

PERIPHERAL PROCESSOR UNIT CHARACTERISTICS

Computation Section

- 12-bit internal word
- Binary computation in fixed-point
- Synchronous internal logic with 27.5-nanosecond clock period

Operating Registers

- 18-bit arithmetic (A) register
- 12-bit program address (P) register
- 12-bit memory read (X) register
- 12-bit instruction (fd) register
- 12-bit working (Q) register

Memory

- 4096 12-bit words of coincident current memory with a parity bit for each 12-bit word (odd parity)
- Organized into two independent banks (2048 words per bank)
- 275-nanosecond read/write cycle time

Input/Output Section

- Eight independent channels (asynchronous)
- Each channel bidirectional (12-bit)

The central processor consists of a CPU, nine functional units, MUX, SCM, and LCM.

Computation is performed by the functional units. Data moves into and out of the functional units through the operating registers (A, B, and X) in the CPU.

The central processor contains three types of internal memory arranged in a hierarchy of speed and size.

1. The instruction word stack contains twelve 60-bit words for issuing of instructions. This register memory is part of the CPU and holds instruction words previously read from central memory. Small program loops can be held in the IWS, thereby avoiding memory references.

2. Central memory (SCM) is used for both program and data storage. The central memory can be either semiconductor memory or small core memory, depending on model. The minimum memory size is 65,536 60-bit words for the semiconductor memory and 32,768 60-bit words for the small core memory. Though the two types of memories are quite different internally, they are nearly identical from the standpoint of system function.

a. Semiconductor memory consists of 16 independent 4K banks for 64K models and 32 banks for 128K models. Since the memory banks are independent of each other, several memory references can take place concurrently within different banks. The memory is bank-phased; that is, sequential addresses lie in different memory banks. This arrangement causes memory cycles to be heavily overlapped during references to sequential locations and results in short, effective cycle times.

b. The small core memory is made up of 16 independent 2K banks for 32K models and 32 banks for the 64K models. Like the semiconductor memory, small core memory is bank-phased to minimize memory conflicts and obtain a short, effective memory cycle time.

3. Large core memory (LCM) is auxiliary storage, connected to central memory by a high-speed data trunk. LCM is used to store information that need not be immediately available to the central processing unit. When needed, programs and data stored in LCM are transferred to central memory where they are directly accessible to the CPU. Large blocks of data can be transferred

rapidly between LCM and central memory. A limited capability exists to transfer data directly between LCM and the CPU; however, programs cannot be executed out of LCM. Minimum LCM size is 256,000 60-bit words. A second 256,000 60-bit words can be added.

The SCM performs certain basic functions in system operation which the LCM cannot effectively perform. These functions are essentially those requiring rapid random access to unrelated fields of data. The first 4K addresses in SCM are reserved for the input/output control and data transfer to service the communication channels to the PPUs. Central processor object programs do not have access to these areas. The remainder of SCM may be divided between fields of program code and fields of data for the currently executing program. A small portion may contain a resident monitor program.

The MUX includes the mechanism to buffer data to (or from) PPUs that are directly connected to the central processor. The PPUs communicate with the central processor over 12-bit bidirectional MUX channels. In the basic system, there are eight channels, one of which is reserved for use by the MCU. Each channel has assembly/disassembly registers to convert 12-bit PPU words to 60-bit central processor words (and conversely). The function of the MUX is to deliver these 60-bit words to SCM for incoming data, read 60-bit words from SCM for outgoing data, and provide the capability to interrupt the central processor for monitor action on the SCM buffer data. Some of the I/O channels are called high-speed channels as opposed to normal channels. High-speed channels transfer data at approximately twice the speed of normal channels. Channels 1 and 10 through 17 (octal) have a SCM buffer area for incoming data and a separate buffer area for outgoing data. Channels 2 through 7 share buffer areas. Each channel also has separate exchange packages for incoming

and outgoing data. The I/O exchange package areas and the buffer areas are permanently assigned in the lowest-order addresses of SCM.

The PPUs are separate and independent computers, some of which reside in the mainframe. Others may be remotely located. A PPU may be connected to the MUX, another PPU, a peripheral device, or a combination of these. PPUs that connect directly to the MUX, whether in the mainframe or remotely located, are termed first-level PPUs. Each PPU has a computation section that performs binary computation in fixed-point arithmetic. A PPU memory provides storage for 4096 12-bit words. This storage is arranged in two independent banks, each with a cycle time of 275 nanoseconds. The PPU instruction set, combined with the high-speed memory and channel flexibility, enables a PPU to drive many types of peripherals without the necessity of an intermediate controller. There are eight input data paths and eight output data paths connecting the PPU to other devices. The PPU input/output facility provides a flexible arrangement for high-speed communication with a variety of I/O devices. The bidirectional channels allow additional PPUs to be added to the system by linking PPU to PPU.

Table 4.2 contains the floating-point instruction timings for the CDC 7600 in multiples of the clock period (CP), which equals 27.5 nanoseconds. $A_i$, $B_j$, and $X_k$ are the address, index, and operand registers, and $(A_i)$ refers to the contents of register $A_i$. K is an 18-bit address or operand.

Table 4.2 CDC 7600 Instruction Timings

Timing Notes:

2. No SCM conflicts or SAS backup caused by SCM conflicts exist.
3. No I/O word request occurs.
4. All operating registers are free.
5. LCM is not busy.
6. All LCM banks have completed previously initiated read/write cycles.
7. The requested LCM bank has completed a previously initiated read/write cycle.
8. The requested operating register(s) is free.
9. If the requested word is in an LCM bank operand register because of a previous reference, the execution time is 3 clock periods.
12. The requested destination register(s) input data path is free during the required clock period.
13. After the instruction has issued to the functional unit, no further delay is possible.
14. The multiply unit is free.
15. The divide unit is free.
18. In models having the semiconductor memory with 256K of LCM, the maximum transfer rate is 32 words per 64 clock periods.
19. In models having small core memory with 256K of LCM, the maximum transfer rate is 32 words per 96 clock periods.

Table 4.2 (Cont'd)

| Mnemonic Code | Instruction Code | Description | Functional Unit | Execution Time (CP) | Timing Notes |
|---|---|---|---|---|---|
| RL | 011jK | Block Copy (Bj) + K Words from LCM to SCM | - | (Bj) + K + 16 | 2,3,4, 5,6,18, 19 |
| WL | 012jK | Block Copy (Bj) + K Words from SCM to LCM | - | (Bj) + K + 12 (13) | 2,3,4, 5,6,18, 19 |
| RX | 014jk | Read LCM at (Xk) to Xj | - | 17 | 5,7,8, 9 |
| WX | 015jk | Write Xj into LCM at (Xk) | - | 3 | 5,7,8 |
| FX | 30ijk | Floating Sum of (Xj) and (Xk) to Xi | Floating Add | 4 | 2,8,12, 13 |
| FX | 31ijk | Floating Difference of (Xj) and (Xk) to Xi | Floating Add | 4 | 2,8,12, 13 |
| DX | 32ijk | Floating Double-Precision Sum of (Xj) and (Xk) to Xi | Floating Add | 4 | 2,8,12, 13 |
| DX | 33ijk | Floating Double Sum of (Xj) and (Xk) to Xi | Floating Add | 4 | 2,8,12, 13 |
| RX | 34ijk | Round Floating Sum of (Xj) and (Xk) to Xi | Floating Add | 4 | 2,8,12, 13 |
| RX | 35ijk | Round Floating Difference of (Xj) and (Xk) to Xi | Floating Add | 4 | 2,8,12, 13 |
| IX | 36ijk | Integer Sum of (Xj) and (Xk) to Xi | Long Add | 2 | 2,8,12, 13 |
| FX | 40ijk | Floating Product of (Xj) and (Xk) to Xi | Multiply | 5 | 2,8,12, 13,14 |
| RX | 41ijk | Round Floating Product of (Xj) and (Xk) to Xi | Multiply | 5 | 2,8,12, 13,14 |

Table 4.2 (Cont'd)

| Mnemonic Code | Instruction Code | Description | Functional Unit | Execution Time (CP) | Timing Notes |
|---|---|---|---|---|---|
| DX | 42ijk | Floating Double-Precision Product of (Xj) and (Xk) to Xi | Multiply | 5 | 2,8,12, 13,14 |
| MX | 43ijk | Form Mask of jk Bits to Xi | Shift | 2 | 2,8,12, 13 |
| FX | 44ijk | Floating Divide (Xj) by (Xk) to Xi | Divide | 20 | 2,8,12, 13,15 |
| RX | 45ijk | Round Floating Divide (Xj) by (Xk) to Xi | Divide | 20 | 2,8,12, 13,15 |

The floating-point, single-precision square-root time (almost 3 microseconds) is sufficiently short with respect to the MOVE time (220 nanoseconds) that we chose LL* decomposition over LDL* decomposition for our implementation on the CDC 7600. A program listing and documentation can be found in Appendix G.

Substituting the times from Table 4.2 into the operation counts of Table 4.1, we obtain the following theoretical timing estimates (in milliseconds):

Time to update the sample covariance matrix per sample voltage vector = $T_u = 10^{-6}\ (495 \cdot N^2)$ 4.2

Time to perform LL* decomposition = $T_D = 10^{-6}(165 \cdot N^3 - 110 \cdot N^2 + 3000 \cdot N + 990)$ 4.3

Time for First or Second Backsubstitution = $T_{FBS} = T_{SBS} = 10^{-6}(495 \cdot N^2 - 220 \cdot N)$ 4.4

Total time to process 2N samples and produce one set of weights = $T_{TOTAL} = 10^{-6}(1155 \cdot N^3 + 385 \cdot N^2 - 2780 \cdot N + 990)$ 4.5

There are, of course, many register loads and stores to and from SCM which are not counted above.

The actual timings of the program were done using a FORTRAN-callable assembly language subroutine called SEC, which returns the number of clock periods used since the beginning of the program. Only the clock periods during which the program was actually executing are counted (i.e., only CPU time is counted). The number of clock periods is returned in the integer variable ICSEC, which is found in the common block /TIME/. Since the clock period is 27.5 nanoseconds, this method of timing is very accurate. Table 4.3 contains the CPU times in milliseconds. The meanings of the various columns are as follows:

1. Number of weights
2. Decomposition - time to factor matrix M into form LL*
3. First Back Substitution - time to solve LT = $\overline{S}$ for T
4. Second Back Substitution - time to solve L*W=T for W
5. Update Sample Covariance Matrix - time to compute and add the outer product of one sample vector to the matrix
6. Total - the time to form the sample covariance matrix using 2N vectors (where N is the number of weights) and solve for W with decomposition and two back substitutions.

It would be interesting to compare the actual times in Table 4.3 with the predicted times of Eqs. 4.2 through 4.5. The form of these equations leads one to believe that a log-log plot would not only be a good means of comparison but would extract the dominating term which controls the growth of the function, since

$$y = ax^b$$

Table 4.3 CDC 7600 CPU Timings in Milliseconds

| N * Weights | Decomposition | First Back Substitution | Second Back Substitution | Update Sample Covariance Matrix | Total Time |
|---|---|---|---|---|---|
| 5 | 0.057 | 0.018 | 0.017 | 0.018 | 0.272 |
| 6 | 0.082 | 0.025 | 0.023 | 0.025 | 0.430 |
| 7 | 0.112 | 0.032 | 0.030 | 0.033 | 0.636 |
| 8 | 0.153 | 0.040 | 0.038 | 0.041 | 0.887 |
| 9 | 0.198 | 0.050 | 0.047 | 0.050 | 1.195 |
| 10 | 0.253 | 0.060 | 0.056 | 0.060 | 1.569 |
| 11 | 0.315 | 0.071 | 0.068 | 0.072 | 2.038 |
| 12 | 0.389 | 0.083 | 0.078 | 0.083 | 2.542 |
| 13 | 0.472 | 0.098 | 0.091 | 0.096 | 3.157 |
| 14 | 0.569 | 0.111 | 0.103 | 0.110 | 3.863 |
| 15 | 0.678 | 0.126 | 0.117 | 0.123 | 4.611 |
| 16 | 0.805 | 0.142 | 0.132 | 0.138 | 5.495 |
| 17 | 0.936 | 0.159 | 0.148 | 0.156 | 6.547 |
| 18 | 1.089 | 0.176 | 0.166 | 0.173 | 7.659 |
| 19 | 1.246 | 0.193 | 0.180 | 0.191 | 8.877 |
| 20 | 1.429 | 0.214 | 0.201 | 0.210 | 10.244 |
| 25 | 2.676 | 0.327 | 0.305 | 0.315 | 19.058 |
| 30 | 4.241 | 0.461 | 0.432 | 0.443 | 31.714 |
| 35 | 6.472 | 0.620 | 0.581 | 0.595 | 49.323 |
| 40 | 9.505 | 0.803 | 0.752 | 0.761 | 71.949 |
| 45 | 13.212 | 1.009 | 0.946 | 0.953 | 100.937 |
| 50 | 17.717 | 1.239 | 1.164 | 1.171 | 137.220 |
| 60 | 29.699 | 1.768 | 1.656 | 1.667 | 233.163 |
| 70 | 46.393 | 2.391 | 2.245 | 2.245 | 365.329 |
| 80 | 68.417 | 3.205 | 2.916 | 2.921 | 541.898 |
| 90 | 96.311 | 3.915 | 3.769 | 3.692 | 768.555 |
| 100 | 130.903 | 4.820 | 4.526 | 4.517 | 1043.649 |
| 110 | 172.760 | 5.817 | 5.464 | 5.446 | 1382.161 |
| 120 | 223.111 | 6.909 | 6.495 | 6.455 | 1785.710 |
| 130 | 282.104 | 8.191 | 7.603 | 7.573 | 2268.177 |
| 140 | 350.537 | 9.370 | 8.894 | 8.766 | 2823.280 |
| 150 | 429.332 | 10.760 | 10.196 | 10.056 | 3467.088 |
| 160 | 522.399 | 12.216 | 11.576 | 11.396 | 4192.910 |
| 170 | 624.772 | 13.779 | 12.929 | 12.851 | 5020.819 |
| 180 | 739.839 | 15.523 | 14.502 | 14.386 | 5948.823 |
| 190 | 865.834 | 17.166 | 16.134 | 15.989 | 6974.953 |
| 200 | 1005.450 | 19.017 | 17.964 | 17.750 | 8142.431 |

implies

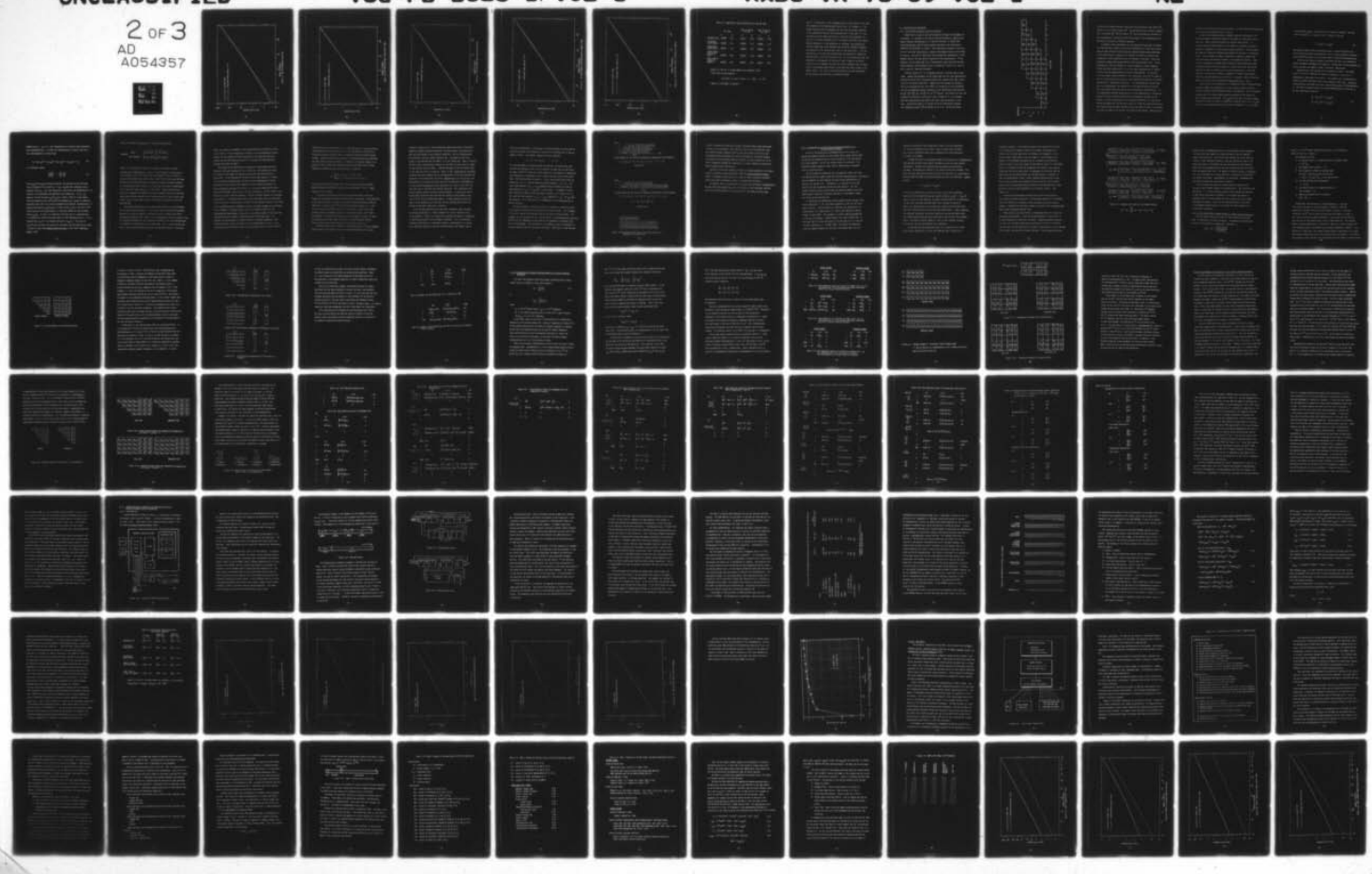
$$\log y = b \log x + \log a$$

and hence that the graph of log y versus log x should be a straight line whose slope equals the exponent b and whose y-intercept is the logarithm of the factor a. Figures 4.5 through 4.9 bear out this suspicion. The solid lines indicate the actual times (CPU milliseconds) and the dashed lines the predicted times as given in Eqs. 4.2 through 4.5. As can be seen, all times produce almost perfectly straight lines. That the actual times increase more slowly than the predicted times, and even become less than the predicted times, would at first seem surprising, since the predicted times did not count overhead and hence would be expected to be consistently less than the actual times. This result can be explained, however, by the CDC 7600's overlap of its functional units: the CDC 7600 has separate hardware for performing multiplications and additions, and can overlap the operation of this hardware. In essence, then, the CDC 7600 is not a purely sequential machine, but has some attributes of parallelism. Overall, the actual and predicted times are in close agreement.

Table 4.4 contains the results of doing a linear least-squares fit to the logarithms of the data in Table 4.3. The least-squares fit was done for all the data taken together (first two columns), for the data for which N, the number of weights, is less than or equal to 30 (middle two columns), and for the data satisfying N > 30 (last two columns). As can be seen from the table, the exponent value B in the expression Time = $A \cdot N^B$ is larger

AD-A054 357 TECHNOLOGY SERVICE CORP SANTA MONICA CALIF F/G 17/9
MULTIDOMAIN ALGORITHM EVALUATION. VOLUME I.(U)
APR 78 W C LILES, J C DEMMEL, I S REED F30602-76-C-0319
UNCLASSIFIED TSC-PD-B525-1-VOL-1 RADC-TR-78-59-VOL-1 NL

2 OF 3
AD
A054357



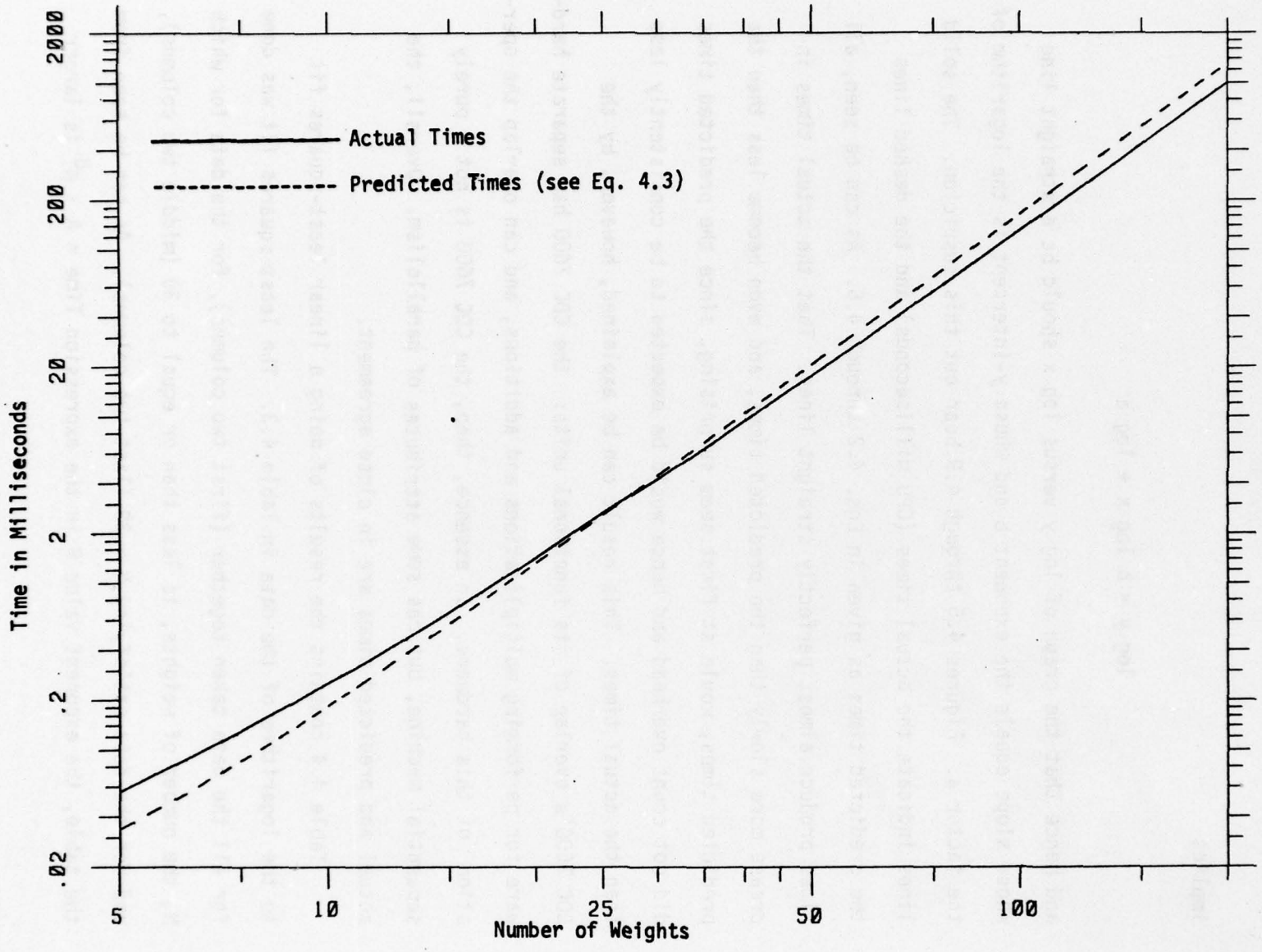



Figure 4.5 CDC 7600 Decomposition Times

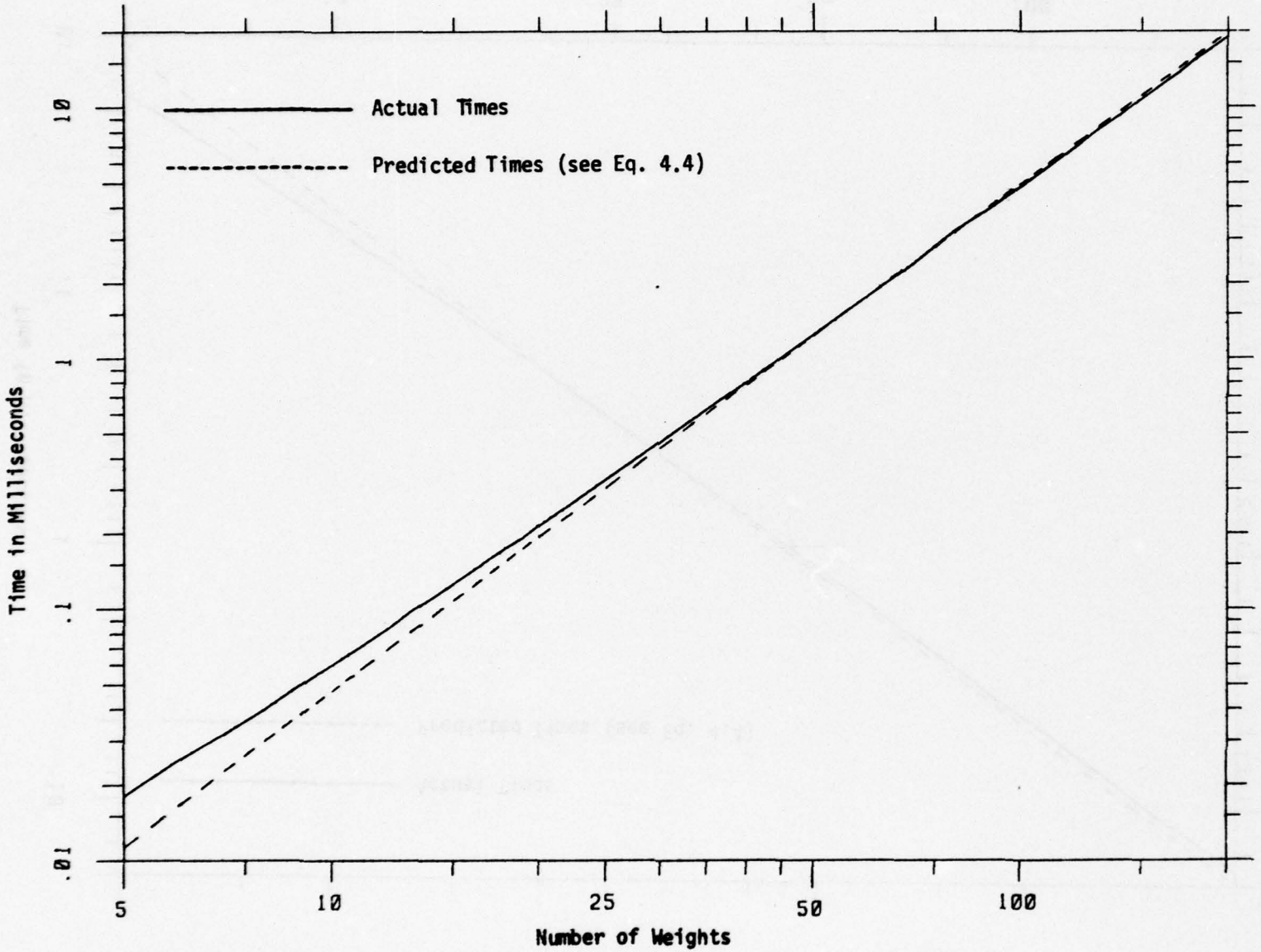



Figure 4.6 CDC 7600 First Back Substitution Times

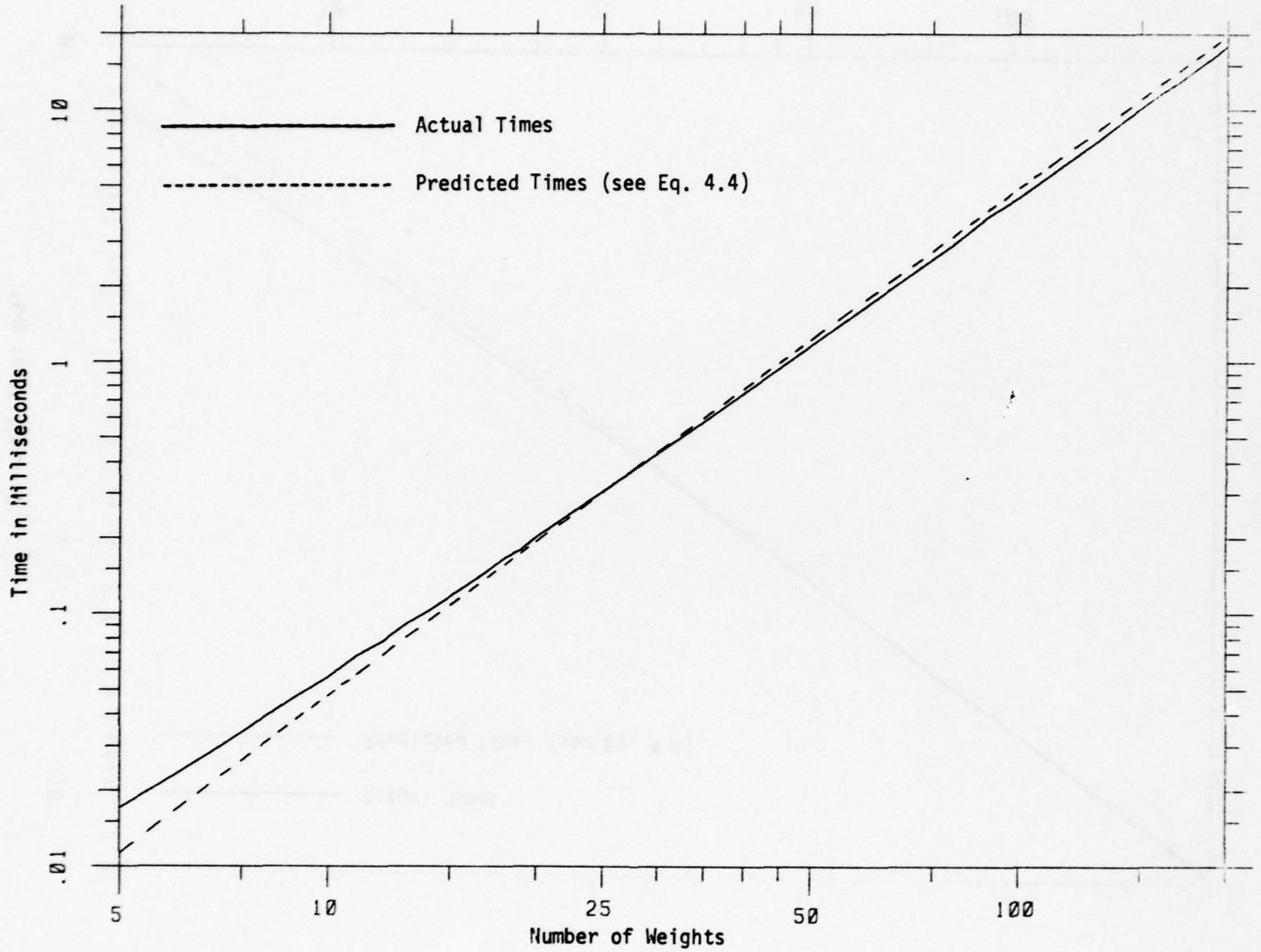



Figure 4.7 CDC 7600 Second Back Substitution Times

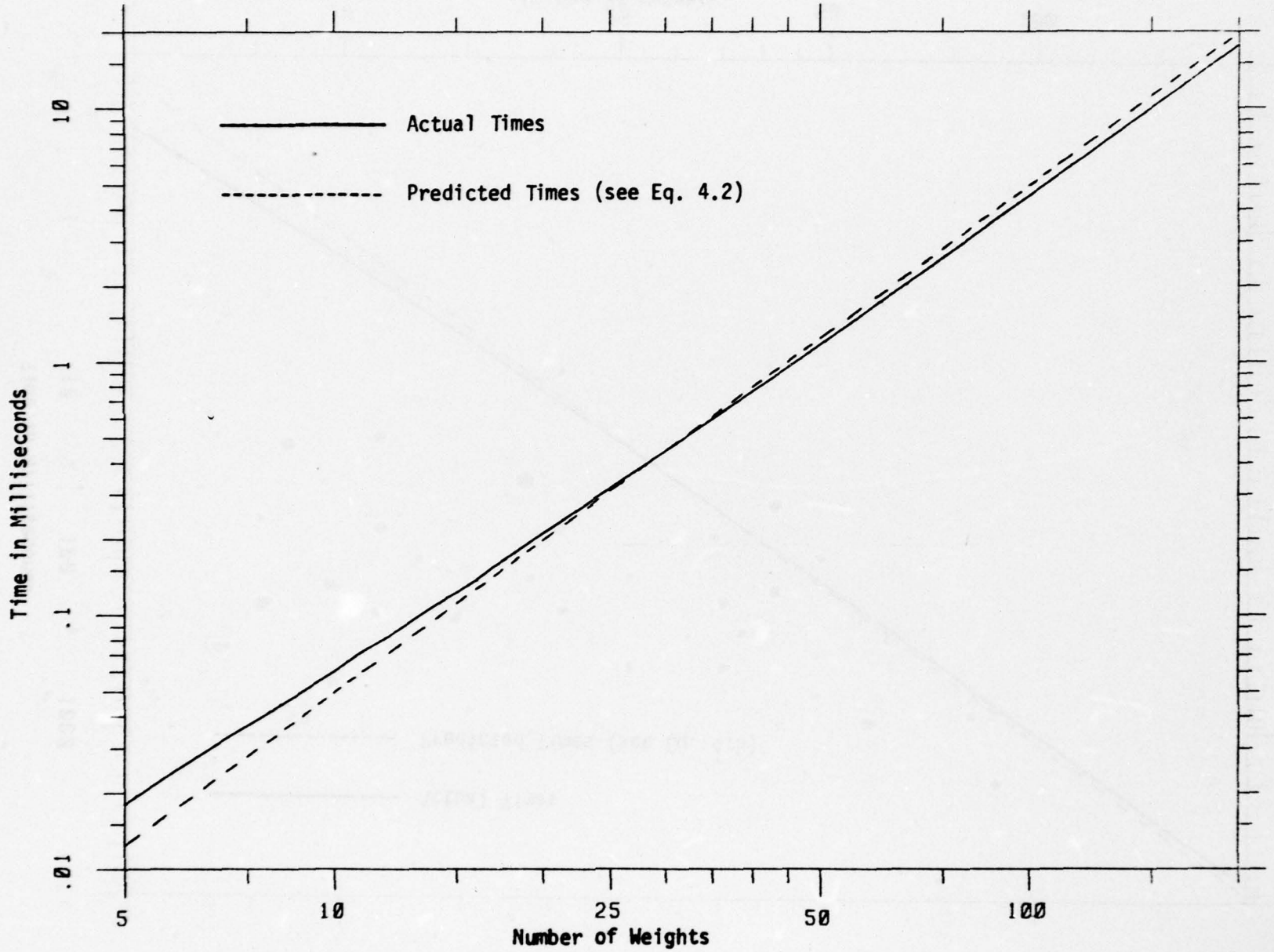



Figure 4.8 CDC 7600 Sample Covariance Matrix Update Times

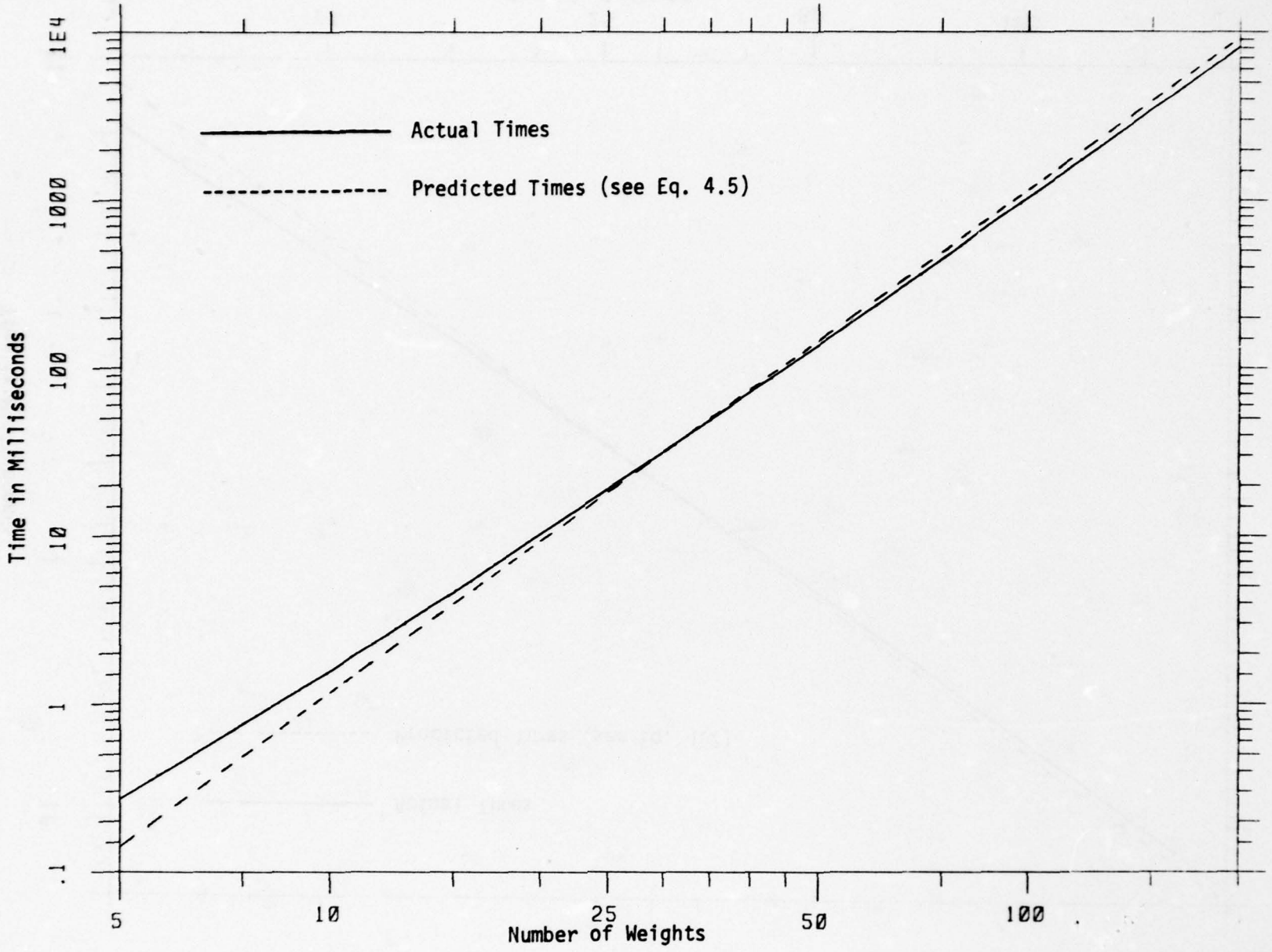



Figure 4.9 CDC 7600 Total Times to Process 2N Samples

Table 4.4 Logarithmic Timing Predictions for the CDC 7600

| | All Data | | Data for Which N ≤ 30 | | Data for Which N > 30 | |
|---|---|---|---|---|---|---|
| | A | B | A | B | A | B |
| Decomposition | .000469 | 2.72 | .001011 | 2.42 | .000210 | 2.90 |
| First Back Substitution | .000736 | 1.91 | .000947 | 1.81 | .000568 | 1.97 |
| Second Back Substitution | .000687 | 1.91 | .000893 | 1.81 | .000524 | 1.97 |
| Update Sample Covariance Matrix | .000780 | 1.88 | .001016 | 1.78 | .000563 | 1.95 |
| Total Time to Process 2N Samples | .002303 | 2.83 | .003581 | 2.65 | .001414 | 2.94 |

(Labels "A" and "B" in column headers are constants in the linear least-squares equation

$$\log (\text{Time}) = B \log N + \log A \quad \text{or} \quad \text{Time} = A \cdot N^B$$

where N is the number of weights.)

when $N > 30$ than when $N \leq 30$, although values in both cases are less than the exponents of the highest-order terms in Eqs. 4.2 through 4.5. The reason that both sets of exponents are less than the highest exponents in the expressions defining the predicted times is that there is a contribution from the lower-order terms in the prediction equations, resulting in an actual exponent which lies between the highest and second-highest exponents. As the dimension, N, increases, the contribution from the lower-order terms decreases and the actual exponents approach the exponents of the highest-order terms in the predictions: 3 for decomposition, 2 for both back substitutions and updating the sample covariance matrix, and 3 for the total time. As can be seen from Figs. 4.5 through 4.9, the exponents of the actual times (slopes of the solid lines) are still somewhat less than the overall exponents (not the exponents of the highest-order terms) of the predicted times (slopes of the dashed lines). This result can be attributed to the 7600 overlapping multiplications and additions, as mentioned before.

## 4.3 VECTOR PIPELINE PROCESSORS

### 4.3.1 Introduction to Vector Pipeline Processors

Pipelining is a type of parallelism used to increase the throughput of a processor. In contrast to the form of parallelism where n identical processors operate simultaneously, in a pipe one processor is broken down into subprocessors, each of which performs one step of the function performed by the processor as a whole. Each subprocessor operates in parallel with all the other subprocessors, producing intermediate results from different inputs. Each set of inputs must pass through each subprocessor, so the speedup results from the parallel operation of the subprocessors. If the processor can be broken down into n subprocessors, each requiring the same time to operate, the throughput can be increased by a factor of n, just as with n parallel identical processors.

Consider Figure 4.10. It illustrates the way a processor may be pipelined. Suppose the processor can be broken down into the five subprocessors instruction fetch (IF), instruction decode (ID), operand fetch (OF), execution (E), and result store (RS). Suppose the five operands $S_1$ through $S_5$ are to be processed and the five results $R_1$ through $R_5$ are to be produced, with the intermediate results contained in each subprocessor being denoted by $S_K^{IF}$, $S_K^{ID}$, $S_K^{OF}$, $S_K^{E}$, and $S_K^{RS}$ for the $K^{th}$ operand. The contents of each subprocessor are shown as a function of time in Figure 4.10. It is assumed that the subprocessors each require one cycle time and operate in lock-step. During cycle time 1, IF fetches the first instruction, produces the intermediate result $S_1^{IF}$, and passes it on to ID. During the second

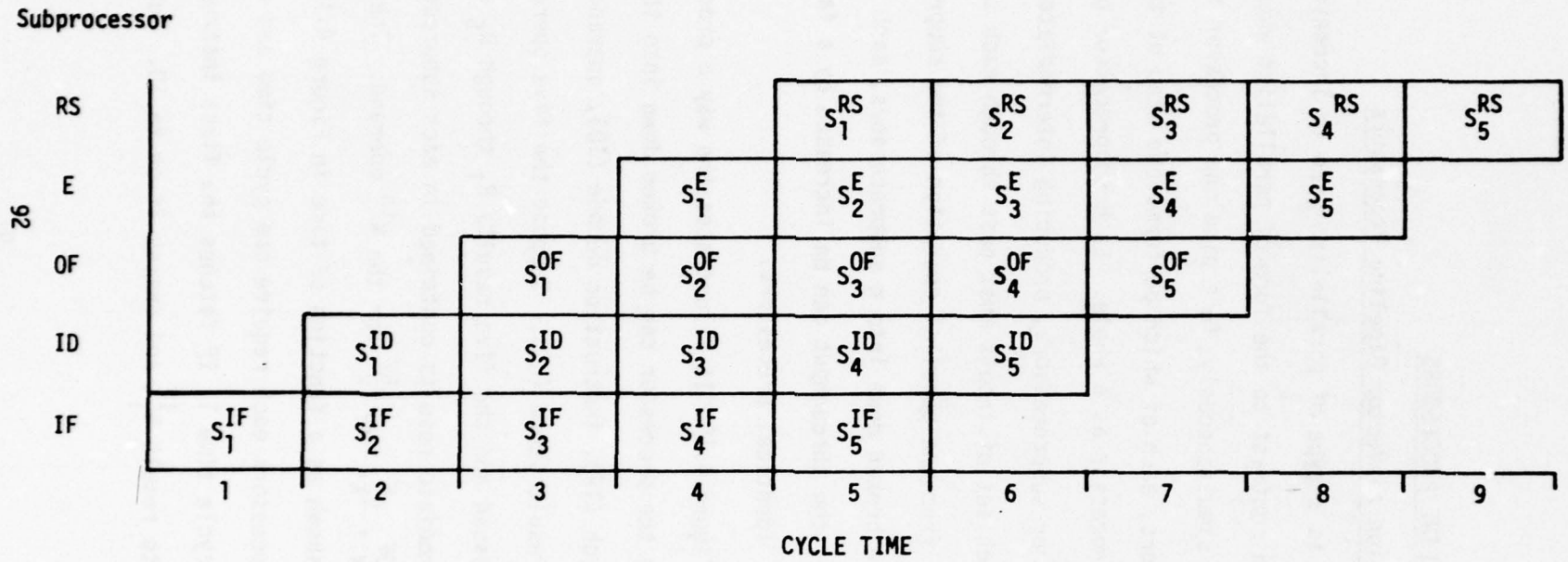



Figure 4.10 Timing Scheme of a Pipeline Processor

cycle time, IF fetches the next instruction and ID processes the results $S_1^{IF}$ passed to it by IF and produces $S_1^{ID}$. During the next cycle time, IF produces $S_3^{IF}$, ID produces $S_2^{ID}$, and OF produces $S_1^{OF}$, and the processing continues in the indicated way. During the fifth cycle time, all five subprocessors are operating in parallel on different operands.

In general, each subprocessor may not require the same time to process its input and pass a result to the next subprocessor; or there may be more than one path through the pipe, bypassing certain subprocessors, depending on the instruction; or there may be feedback of results to previous subprocessors; or the subprocessors may not operate in lockstep. These complications produce many design problems which must be dealt with. If the subprocessors require different or variable amounts of time to operate, either some delay must be built in to the faster subprocessors or buffers to save inputs are needed before the slower subprocessors. If the pipe must reconfigure itself according to the instruction, care must be taken so that an instruction does not cause a reconfiguration and prevent the correct processing of a downstream instruction. If a configuration involves feedback to a subprocessor, the scheduling must be done to avoid collisions (two instructions trying to use the same subprocessor simultaneously). If the subprocessors operate asynchronously and one subprocessor is delayed because of an error condition, and another instruction is allowed to pass through, it must be determined if the bypassed instruction was required to produce an operand for the new instruction, or if the two instructions write their results into the same location so that the result of the first instruction must be stored first and then the result of the second. There are also

problems of branching and interrupt handling. All these design considerations are discussed by Ramamoorthy and Li [1977].

So far we have been discussing a general pipeline architecture, and there have been no conditions placed on the types of instructions fed to the pipe. Vector pipelining results in several simplications. In a vector pipe the same operation is performed on a sequence of operands located in sequential or evenly spaced memory locations. This eliminates the need for instruction fetching and decoding in the pipe, or reconfiguration. For the common arithmetic operations of addition, subtraction, multiplication, and division, feedback is not generally required. Since the same operations are always being performed, each subprocessor takes a constant amount of time to produce its results and so the pipe may be designed to have its subprocessors function synchronously. Since the operands are required to reside in sequential or evenly spaced memory locations, operand fetching is simplified. In general, vector processing lends itself very well to pipelining [Ramamoorthy and Li, 1977].

What we would like to develop is a theoretical means of comparing implementation speeds and complexities on vector pipeline machines without having to refer to any individual machine's architecture. This is possible to some extent for vector pipeline machines because of the common way their timings may be analyzed. Let $t_{op}(X)$ be the time in seconds between consecutive issues of results of operation X from the pipe, and let $t_{su}(X)$ be the time in seconds for the first result of operation X to issue from

the pipe minus $t_{op}(X)$. The subscript "su" stands for startup. Then the time in seconds, T, to process n sets of inputs is given by

$$T = t_{su}(X) + n\, t_{op}(X) \quad . \qquad 4.6$$

The equation implies that the operation count is linearly related to the time required to perform the operations, as in a purely sequential processor, so that operation counts are a potential means of comparing implementations.

There are four factors, however, which make the naive use of operation counts to compare implementations dangerous: startup time, the way operands are fed to the pipe, the number of pipes, and special instructions.

Startup time is the constant term in Equation 4.6 above. The larger $t_{su}(X)$ is, the less important the number of operations, n, will be in determining T, particularly for small n. For example, let Implementation A require $m_A$ vector operations with a total operation count of $n_A$. Similarly, let $m_B$ and $n_B$ be the number of vectors and operation counts for Implementation B. The times in seconds required to perform Implementations A and B are $T_A$ and $T_B$, respectively, given by:

$$T_A = m_A\, t_{su}(X) + n_A\, t_{op}(X)$$

$$T_B = m_B\, t_{su}(X) + n_B\, t_{op}(X) \quad . \qquad 4.7$$

Assume that $n_A < n_B$, i.e., that Implementation A requires fewer operations than Implementation B. In order that Implementation A require less time than Implementation B, we must have

$$T_A = m_A\, t_{su}(X) + n_A\, t_{op}(X) < m_B\, t_{su}(X) + n_B\, t_{op}(X) = T_B \qquad 4.8$$

or, rearranging terms,

$$\frac{t_{op}(X)}{t_{su}(X)} > \frac{m_A - m_B}{n_B - n_A} \quad . \qquad 4.9$$

Since $t_{op}(X)$ and $t_{su}(X)$ are both positive, the fraction on the left-hand side of Equation 4.9 is positive. If $m_A < m_B$, then the right-hand side is negative (since $n_A < n_B$), the inequality is satisfied, and Implementation A is superior, as predicted by the operation count comparison. If $m_A > m_B$, however, and the fraction $t_{op}(X)/t_{su}(X)$ is small, there is a good chance that the inequality will not be satisfied whenever $n_A < n_B$, and so comparing operation counts will not correctly determine the superior implementation. In general the fraction $t_{op}(X)/t_{su}(X)$ is small, since $t_{su}(X)$ is the sum of the times required by the subprocessors in the pipe, except for the one determining $t_{op}(X)$. In fact, in a maximally efficient pipe of n subprocessors, all subprocessors require the same amount of time [Ramamoorthy and Li, 1977], implying that $t_{su}(X)$ is an integral multiple of $t_{op}(X)$. The values of $t_{op}(X)/t_{su}(X)$ and their reciprocals for the CRAY-1 and CDC STAR-100 are shown in Table 4.5 [CDC STAR Hardware Reference Manual, 1975; CRAY-1 Reference Manual, 1976].

Table 4.5 Values of $t_{op}(X)/t_{su}(X)$ (and Their Reciprocals)

| COMPUTER: | + | * |
|---|---|---|
| CRAY-1 | .125 (8.0) | .111 (9.0) |
| CDC STAR-100 | .007 (140.0) | .014 (70.0) |

Hence, it is necessary to consider the number of startups required evaluating an implementation. In fact, if the machine's startup time is long enough and its scalar operations fast enough, it may be faster to process short vectors by a scalar loop. Even if the machine uses the same hardware to do scalar and vector operations, the extra overhead involved in configuring the pipe to process a vector may still make it advantageous to use loops for short vectors as in the CRAY-1. When programming in a high-level language like FORTRAN, the extra overhead incurred may override these subtle effects and the vector operation may still appear faster than scalar loops for very short vectors, as has been our experience with the CDC STAR-100.

The startup time may not only depend on the operation to be performed, but on the previous operation. If one pipe is used to perform two separate operations, it must in general reconfigure itself to perform them, which increases the startup time. Hence, if an implementation can be arranged to have its operations grouped, all the additions followed by all the multiplications, for example, it would decrease the startup time. This is true only when one pipe is used to perform both operations; for multipipe machines, discussed

below, the opposite arrangement, alternating additions and multiplications, may be faster. But for single-pipe machines, of two implementations which require the same number of vector operations of the same type and same length, one may definitely be superior if it groups the same operations together more than the other.

The second factor which makes it necessary to look beyond operation counting is the way operands are fed to the pipe. For example, there may be a limit to the number of operands which can be fed into the pipe in one stream. If the operands are fetched directly from memory or if there is a fast buffer fetching data from memory while simultaneously feeding the pipe, the only limit on the length of a vector may be the number of bits allocated to describe it (16 bits, or a vector of length 65,535 words on the CDC STAR-100, which is not a serious limitation). If the vector is contained in a vector register, however, it may be of quite limited size (64 words on the CRAY-1). This means that longer vector operations must be broken down into shorter ones, involving more startup time and register loading, so it would be to an implementation's advantage to limit its vector lengths to this maximum size. Feedback is another reason a pipe may be better suited for one implementation than another. Being able to feed back results of the pipe into the operand stream results in increased versatility of the pipe. For example, the dot-product is a common operation in matrix manipulation and involves summing a vector of numbers stored sequentially in memory. If it were possible to divide this vector into two pieces, feed the pieces into an

adder pipe and pair up the results, and feed them back in, the sum could be accomplished much more quickly then by scalar processing or by having to restartup the pipe after each vector of paired-up partial sums had passed through. Assuming that the pipe consists of m subprocessors, each requiring the same amount of time in seconds, t, and that n, the vector length, is a power of 2 (for simplicity), Ramamoorthy and Li have shown that the time required to perform the sum, given feedback, is given by

$$T = \begin{cases} t[n + m \log_2 m - 1] & \text{if } n \geq 2m \\ t[n/2 + m \log_2 n - 1] & \text{if } 2 \leq n \leq 2m \end{cases} .$$

If the pipe had to be restarted after each sequence of partial sums had been processed, an additional $\log_2 n$ startups requiring $t \cdot (m-1) \cdot \log_2 n$ seconds would be needed.

The number of different operand storage patterns which a pipe can accept can help decide between one implementation and another. Many pipes will accept only data that is stored in consecutive memory positions, making it impossible to process every $n^{th}$ data word efficiently, where $n \geq 2$. This is true of the CDC STAR-100 and CRAY-1, for example. Other pipes allow constant address increments between data words other than 1, the TI ASC for example. This would make operating on the columns of a matrix as easy as operating on the rows (when the matrix is stored rowwise), whereas the first kind of pipe mentioned could operate only on the rows.

The third factor that, in addition to operation counts, affects implementation choice is the number, types, and interrelationships of the pipes.

Instead of having one or a few multipurpose pipes which have to reconfigure themselves between different operations (discussed above), the machine may have one pipe devoted to each possible operation: one for floating-point multiplication, one for integer addition, etc. An example of the first system is the CDC STAR-100; the CRAY-1 is of the second type. Note in Table 4.5 the much larger startup times (relative to the times for one result to issue) for the STAR-100 than the CRAY-1. Another advantage of having several pipes is that they may be run in parallel. Hence, if one implementation alternates its operations while another has all the operations of one type preceding the operation of the second type, the first implementation will run faster since the machine will be able to process each pair of operations simultaneously; for the second implementation, most operations will have to wait until the pipe they need is done with the previous operation. This multipipe architecture presents several difficult design problems: in particular, how to handle memory conflicts when two pipes are trying to simultaneously read from or write into the same memory banks. If these problems can be overcome, a great speedup can be realized, not just through parallelism, but through second-order pipelining or chaining.

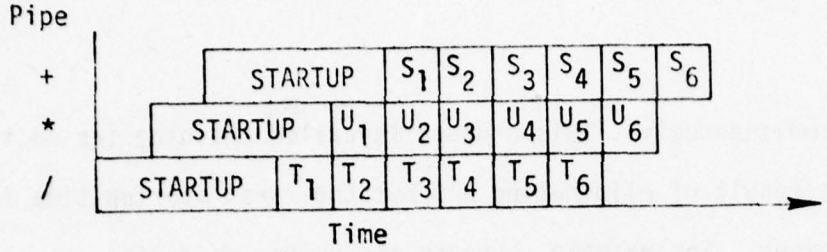
So far we have been discussing low-level pipelining, where the operation being pipelined is a simple arithmetic or logical function of one or two operands. There is no reason not to expand the concept of pipelining to complicated functions of any number of operands. This is possible by considering each pipe in a multipipe system as a subprocessor in a larger pipe, where the results of one pipe are fed directly into another pipe as

they run simultaneously. This process is called chaining (as on the CRAY-1) and has the result of eliminating all but the first startup time in the sequence of pipes. For example, suppose the vector operation

$$S_i = A_i + B_i * (C_i/D_i) \qquad i = 1,\ldots,6$$

is to be performed, and separate pipes for addition, multiplication, and division are available, each of which requires the same time for one result to issue. Figure 4.11 shows the timing scheme for this sequence of operations and indicates the intermediate result issuing from each pipe at each cycle time. The division pipe calculates $T_i = C_i/D_i$, the multiplication pipe calculates $U_i = B_i * T_i$, and the addition pipe calculates $S_i = A_i + U_i$. The total time required is $t_{su} + 8\ t_{op}$; whereas, without chaining, the time required would be $3\ t_{su} + 18\ t_{op}$, which is much slower. If feedback is available, then we can chain similar operations together, as in Figure 4.12. In this case the time required is $t_{su} + 12\ t_{op}$, compared to $3\ t_{su} + 18\ t_{op}$ without chaining. It is clear that it is advantageous for an implementation to utilize chaining.

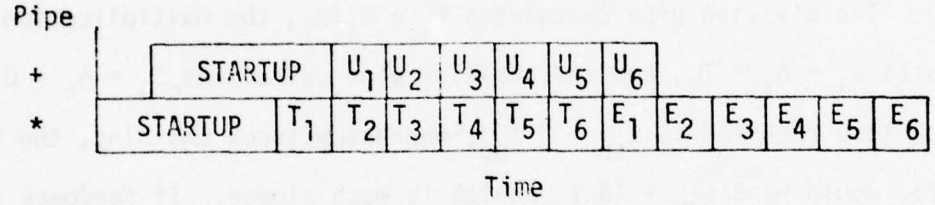
The last category of factors to consider in addition to operation counts is the special instruction set available on the pipe or pipes. This category is a kind of catchall and overlaps some of the factors discussed above. For example, not all machines may have division pipes, so an implementation requiring division of the components of one vector by the components of another would be at a disadvantage. The availability of a command to sum the components of a vector makes the use of dot-products efficient. Being able to have the data




Timing Scheme for the Following Sequence of Operations Using Chaining:

$$T_i = C_i/D_i,\ U_i = B_i * T_i,\ S_i = A_i + U_i,\quad i = 1,\ldots,6$$

Figure 4.11




Timing Scheme for the Following Sequence of Operations Using Chaining:

$$T_i = C_i * D_i,\ U_i = B_i + T_i,\ E_i = U_i * A_i,\quad i = 1,\ldots,6$$

$$E_i = (B_i + (C_i * D_i)) * A_i$$

Figure 4.12a

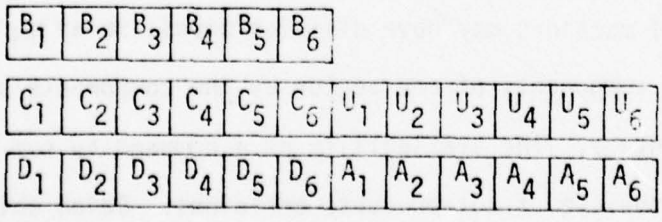



Figure 4.12b Storage Scheme Needed to Perform Sequence of Operations in Figure 4.12a

stored in constant but non-unity address increments makes column operations as easy as row operations. Some machines, such as the CDC STAR-100, have bit control vectors available which can be used either to turn on or off the operation being performed for selected elements of the vector (making non-unity or even nonconstant address increment handling possible) or to insert 0 components between the components of a vector (making it possible to handle sparse vectors efficiently).

The CDC STAR-100 has hardware commands to evaluate polynomials and transpose 8 x 8 matrices using pipes [CDC, STAR-100 Hardware Reference Manual, 1975]. In general, there may be any number of exotic functions available which could make an otherwise inferior implementation efficient.

In general, while it is possible to roughly order different implementations by operation counts comparisons, the architecture of each individual machine must be scrutinized before a final decision for that machine can be made.

### 4.3.2 Implementations of Algorithms to Determine Weights For A Vector Pipeline Processor

The basic algorithms whose implementations we will discuss for determining adaptive weights are the inverse matrix update method (IMU), and forming the covariance matrix M and solving $MW = \overline{S}$ for the weights W by direct methods. We will not discuss loops or solving $MW = \overline{S}$ by iterative methods because of their slow or indeterminate convergence rates as discussed above.

The basic means of comparison will be operation counts, but since this method can only roughly order the different implementations, the analysis must be repeated for each computer and each system configuration, as discussed in Section 4.3.1. Therefore, it is necessary to list the restrictions and assumptions we make for each analysis. For this general analysis, we have tried to make the fewest assumptions possible about special functions or features available, or the amount, format, and timing of data availability.

In particular, the operations we have counted include startups (SU), vector operations (V OP), and scalar operations (S OP) for the arithmetic functions: multiplication (*), additions and subtractions ($\pm$), reciprocation (/), square roots ($\sqrt{\ }$), moving data (MOVE), and summing vectors of data (SUM). Each category is further subdivided depending on the type of operands: complex-complex (CC), real-complex (RC), or real-real (RR) for binary operations (*,$\pm$) and complex (C) or real (R) for unary operations (/, $\sqrt{\ }$, MOVE, SUM). Eventually the operations involving complex numbers (CC, RC, and C) are broken down into real

operations according to the assumptions listed in the next paragraph. All vectors operations are at least 2 sets of inputs long, and there are no scalar operations which can be combined into vector operations of length 2 or greater.

In order to limit to a reasonable number the variations of implementations that are well-suited to most vector processors, we have to make several assumptions. We must first consider the way complex multiplications are done. As discussed in Section 3.2.2, there are two basic methods: one requiring four real multiplications and two real additions (or subtractions), and one requiring three real multiplications and five real additions. Complex

Letting

$$T = t_{su}(X) + n \cdot t_{op}(X) \quad ,$$

we have shown before that not only does the $n \cdot t_{op}(X)$ term contribute less to the first algorithm than the second as long as $t_{op}(*) < 3 \cdot t_{op}(\pm)$ (which is true of most machines, the CRAY-1 and CDC STAR-100 in particular), but the first algorithm requires fewer total startups (6 versus 8). Hence, our logical choice is to convert one complex multiplication into four real multiplications and two additions/subtractions. Complex additions are obviously equivalent to two real additions, and moving complex data and summing complex data are also equivalent to two of their real counterparts. Multiplying complex numbers by real or pure imaginary numbers involves two real multiplications and no additions.

Moving data has been considered since it is sometimes done at the same speed as additions, as on the CDC STAR-100, and is hence not a

negligible quantity. As discussed above, the time required to sum data is not necessarily linearly related to the number of operands to be summed (when using an adder pipe and feedback, for example); so it is not universally applicable to use the number of the operands to be summed in all different SUM operations as a measure of how long the sums will take, but we will assume for the analysis that it is sufficient. Since it is possible to sum the components of a vector in many ways by collapsing the sum by adding pairs of components of the vector with an adder pipe, we have also assumed that the SUM function is optimal in that no precollapsing is done. Hence, complex multiplications to produce a complex vector to be summed break down into four real multiplications only, the additions being performed by two SUM operations, each one having twice as many components as the original vectors. See Figure 4.13.

We have also assumed that operands can be fed to the pipe only from sequential memory locations; no simultaneous operations of different pipes are allowed, no chaining is done, and there are no bit control vectors. Attention will be drawn, however, to parts of the implementations where these techniques might be used.

There are very few restrictions or assumptions made with respect to the system configuration. Since the covariance matrix is Hermitian, it is only necessary to store half the matrix. When the problem dimension is large, this storage scheme may be necessary to fit the entire matrix into the core, or the fast part of the core if the machine's core is divided into a small, fast part and a large, slow part. This storage scheme can

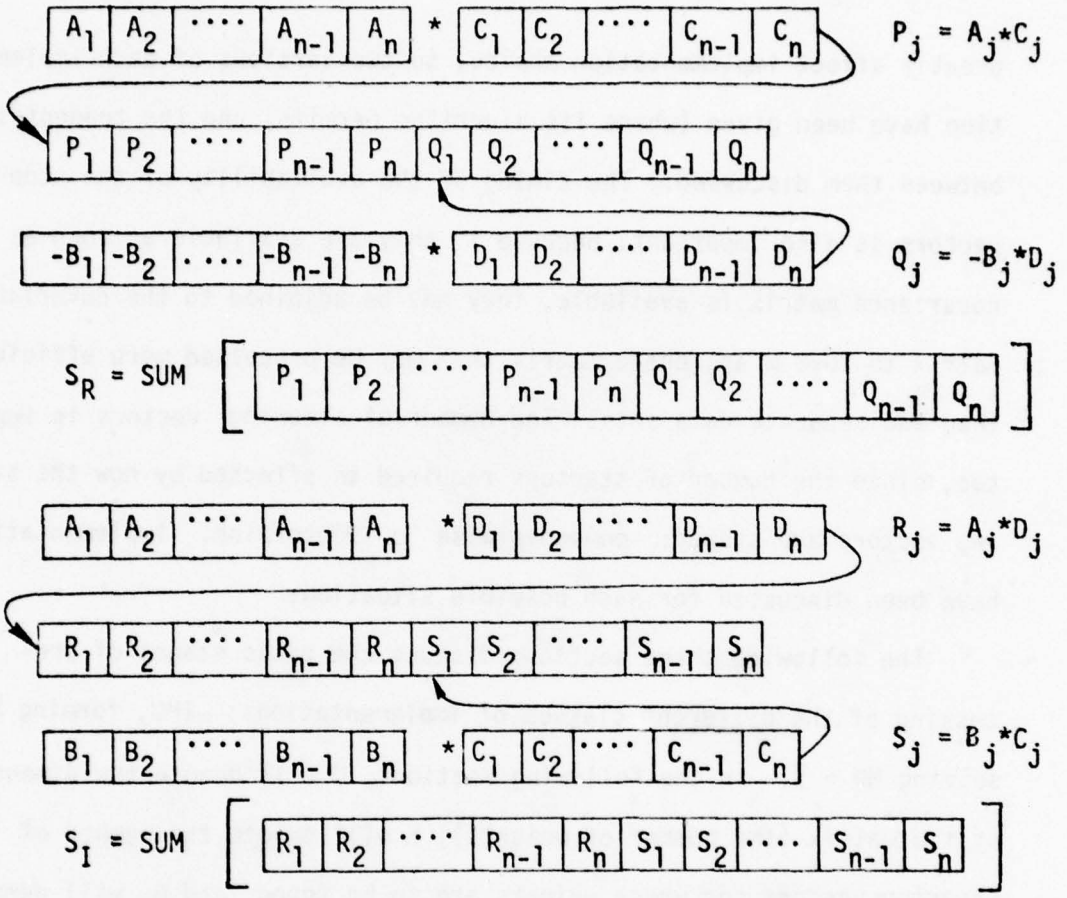



Figure 4.13 Summing the Product of Two Complex Vectors

$$S_R + iS_j = \sum_{j=1}^{n} (A_j + iB_j) * (C_j + iD_j)$$

greatly affect implementation choice, so two versions of each implementation have been given (where its algorithm permits) and the tradeoffs between them discussed. The timing of the availability of the steering vectors is also important, because if they are available as soon as the covariance matrix is available, they may be adjoined to the covariance matrix to form an augmented matrix that may be processed more efficiently than two separate data sets. The number of steering vectors is important. too, since the number of startups required is affected by how the steering vectors are stored: componentwise or vectorwise. Implementations have been discussed for each possible situation.

The following three sections discuss the basic stages of processing of the different classes of implementations: IMU, forming M, and solving $MW = \bar{S}$. In the following sections, N will denote the dimension of the matrix (the number of weights), K will denote the number of steering vectors for which weights are to be found, and $N_s$ will denote the number of samples. M will be the covariance matrix, S the array of conjugated steering vectors, X the vector of input voltages, and W the vector of weights.

4.3.2.1 The Inverse Matrix Update Method on a Vector Pipeline Processor

IMU has several features, besides its high operation count, that make it unsuitable for vector processors. The equation describing IMU, as given in the section on mathematical techniques, is

$$M_{i+1}^{-1} = M_i^{-1} - \frac{M_i^{-1}\bar{X}_{i+1}X_{i+1}^T M_i^{-1}}{1+X_{i+1}^T M_i^{-1}\bar{X}_{i+1}} \quad , \qquad 4.10$$

where $X_i$ is the $i^{th}$ sample voltage vector and $M_i$ is the covariance matrix of the first i samples.

This equation involves

1) one multiplication of a complex matrix by a complex vector

   $Y = M_i^{-1} \overline{X}_{i+1}$

2) one complex outer product $Z = YY^T$

3) one dot-product without an imaginary part;

   since $M_i^{-1}$ is Hermitian positive definite

   $\gamma = X_{i+1}^T Y = X^T{}_{i+1} M_i^{-1} \overline{X}_{i+1}$ is real and positive

4) one real addition and reciprocation

   $\sigma = 1/(1+\gamma)$

5) one multiplication of a complex matrix by a

   real constant $T = \sigma Z$

6) adding one complex matrix to another,

   $M_{i+1}^{-1} = M_i^{-1} + T$ .

Assume first that the matrix is stored packed, i.e., that the first row of the matrix is stored starting in column 1; that the second row is stored starting with the element in column 2; and that, in general, the $j^{th}$ row is stored starting with the element in column j, for a total of N(N+1)/2 complex locations, or two arrays, one for the real parts and one for the complex parts, each of length N(N+1)/2. This storage method is shown schematically in Figure 4.14. Here, as in other figures, solid lines separate vectors, and dotted lines separate components of vectors. Since the matrix is Hermitian, this rowwise storage method is equivalent to a column-wise method, except the signs of the imaginary parts are reversed. This method cuts storage in half, and also cuts the maximum possible number of multiplications

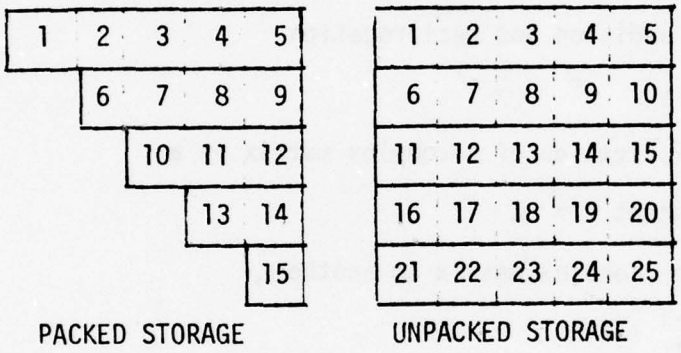



Figure 4.14 Storage Methods for Hermitian Matrices

in steps 2, 5, and 6 in half. The difficulty lies in computing the dot-products in step 1, because the elements of the matrix which have to be multiplied by the components of the vector are not located in sequential locations except for the first row. When N = 5, for example, in order to calculate the third coordinate of the product vector Y, we must multiply and sum the 5 numbered pairs of elements in Fig. 4.15a. Elements 1, 2, and 3 of the matrix are not in sequential or even evenly spaced memory locations, making the use of a pipe difficult and outside the domain of our assumptions mentioned above. If bit control vectors were available, this product and sum could be accomplished by using a bit control vector having a value of 1 in locations corresponding to values to be included in the sum, and zeros elsewhere. This approach would involve passing the whole matrix through the pipe, since the pipe must match up each element of the matrix with a bit and decide whether to use it, and hence the time required would be proportional to the length of the whole matrix, not just the vector inside it.

If the matrix is not stored packed, there are two possibilities. If all locations are filled with correct data, then the dot-products are straightforward, as in Figure 4.15b, but twice as many multiplications and additions are required in steps 2, 5, and 6, as when the matrix is packed. It is also possible not to fill the entire matrix with correct data, but just use the space so that elements in a column are separated by constant address increments, as in Figure 4.15c. Elements 1, 2, and 3 in $M_i^{-1}$ are separated by constant address increments, as are elements 3, 4, and 5,

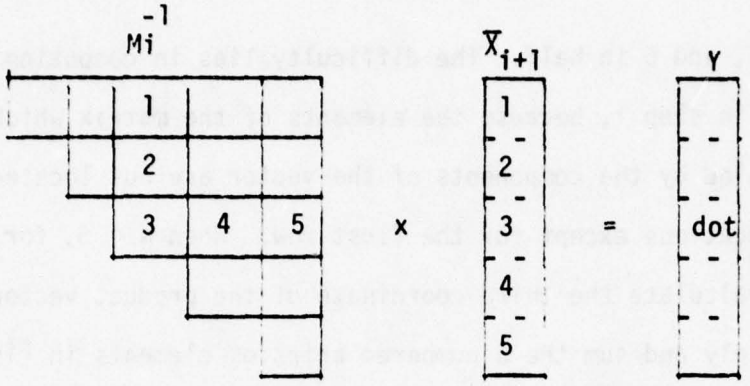



Figure 4.15a Multiplying a Packed Matrix by a Vector

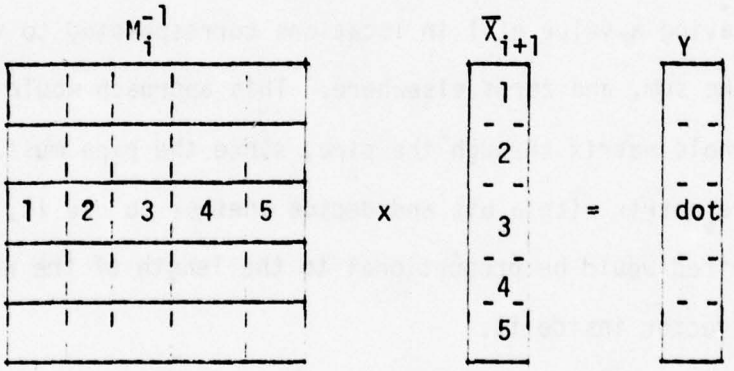



Figure 4.15b Multiplying an Unpacked, Filled Matrix by a Vector

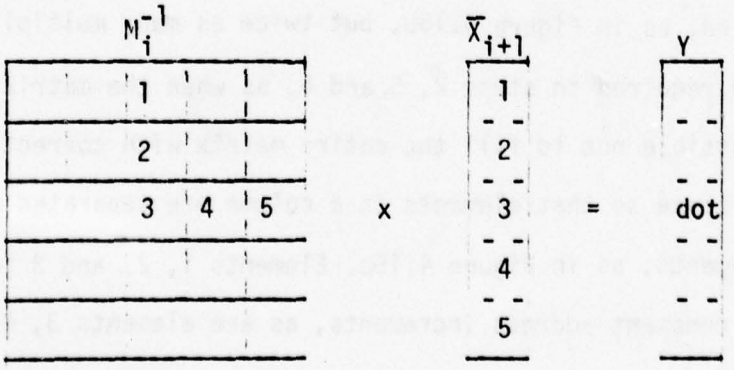



Figure 4.15c Multiplying an Unpacked Half-Filled Matrix by a Vector

so that on a machine which allows non-unity constant address increments, the whole product can essentially be vectorized and pipelined. Steps 2, 5, and 6 require only as many operations as the packed version of the algorithm, so this method seems best in terms of operation counts and suitability for pipelining.

Even in this last case, however, the operation counts are larger than those for the direct methods of the next sections, even assuming only half matrices need to be used in steps 2, 5, and 6 and that the only overhead accrued by the dot-product is four startups for the multiplications and two for the sums. Table 4.6a contains the operation counts for one iteration of IMU as described in steps 1 through 6 above. All complex operations have been decomposed into their constituent real operations.

This algorithm must be repeated for each new sample, and finally $M_i^{-1}$ must be multiplied by the steering vector or vectors to form the weights. Table 4.6b contains the total operation counts as a function of the number of samples and steering vectors.

| | SU | V OP | S OP |
|---|---|---|---|
| * | 8N | $7N^2+5N-4$ | 4 |
| ± | 2N | $2N^2+2N-2$ | 1 |
| / | 0 | 0 | 1 |
| SUM | 2N+1 | $4N^2+2N$ | 0 |

Table 4.6a Number of Real Operations for 1 Iteration of IMU

| | SU | V OP | S OP |
|---|---|---|---|
| * | $(8N)N_s+(4N)K$ | $(7N^2+5N-4)N_s+(4N^2)K$ | $4N_s$ |
| ± | $(2N)N_s$ | $(2N^2+2N-2)N_s$ | $N_s$ |
| / | 0 | 0 | $N_s$ |
| SUM | $(2N+1)N_s+(2N)K$ | $(4N^2+2N)N_s+(4N^2)K$ | 0 |

Table 4.6b Number of Real Operations for IMU to Form K Sets of Weights from $N_s$ Samples

#### 4.3.2.2 Calculating the Sample Covariance Matrix on a Vector Pipeline Processor

We recall the equations describing sample covariance matrix computation (with unit weights on each outer product):

$$M_i = \sum_{j=1}^{i} \overline{X}_j X_j^T \qquad 4.11$$

or

$$M_{i_{k\ell}} = \sum_{j=1}^{i} \overline{X}_{jk} X_{j\ell} \quad , \qquad 4.12$$

where

$X_j$ is the $j^{th}$ sample vector, $X_{jk}$ is its $k^{th}$ component,

$M_i$ is the sample covariance matrix of the first i sample vectors,

and $M_{ik\ell}$ is its $(k,\ell)^{th}$ component.

As with IMU, there are two sets of implementations for computing $M_i$, depending on the storage scheme used: packed or unpacked (see Figure 4.14). In the packed scheme we have only $N(N+1)/2$ complex components to compute, whereas with the unpacked version we may have $N^2$ complex components. These multiplications and the corresponding additions to update the matrix are essentially unavoidable, so the major difference between implementations will be in the overhead involved.

There are three basic implementations, each of which has both a packed and unpacked form. Implementation 1 is the straightforward method of multiplying the input vector transpose $X_j^T$ by the conjugate of its $i^{th}$ component $\overline{X}_{ji}$ with a complex vector multiply and adding the product to

the $i^{th}$ row of the sample covariance matrix with a complex vector add. Note that we know the diagonal elements will always be real since

$$M_{i_{kk}} = \sum_{j=1}^{i} \overline{X}_{j_k} X_{j_k} = \sum_{j=1}^{i} |X_{j_k}|^2 \quad ,$$

so we need not calculate any imaginary term for that element. In the unpacked case, however, where the entire $i^{th}$ row is calculated (not just from the $i^{th}$ column right), taking advantage of this one a priori known value would require dividing the two real vector multiplications and additions needed to calculate the imaginary parts of each row into two parts, doubling the number of startups. Thus if we insist on saving two vector multiply times and two vector add times,

$$2\ t_{op}(*) + 2\ t_{op}\ (\pm) \quad ,$$

we will lose 4 startup times,

$$2\ t_{su}(*) + 2\ t_{su}\ (\pm) \quad ;$$

and since, in general, $t_{op} < t_{su}$, it is faster to perform the extra operations. The operation counts for Implementation 1 are in Table 4.7a; corresponding code is given in Appendix A.

Implementation 2 attempts to save on startup times by "stretching out" all the data to be multiplied and added so all the multiplications and additions can be done in one operation. This requires a great deal of extra storage and data movement and would need a machine for which $t_{op}$ (MOVE) and $t_{su}$ (MOVE) were small compared to $t_{su}$ (*) and $t_{su}$ (±).

Fig. 4.16 shows the way each sample vector $X = X_R + iX_I$ would have to be restored in the case N=4 for this implementation. If $X'_R$, $X''_R$, $X'_I$, and $X''_I$ are the new vectors in Figure 4.16, we can update M with the following vector operations:

$$M_R = X'_R * X''_R + X'_I * X''_I$$

$$M_I = X'_R * X''_I - X'_I * X''_R \quad .$$

The operation counts are given in Table 4.7b and corresponding code in Appendix A.

The third implementation we consider assumes $N_s$ sample vectors are available in core and stored componentwise as in Figure 4.17a. The matrix is now updated by forming the inner product of the appropriate components of the sample vectors, as expressed in Equation 4.12. Hence, one complex inner product of length $N_s$ components must be performed for each element of the matrix N*(N+1)/2 in all. This implementation would need an extremely fast SUM function to compete with the other methods. Its operation counts are in Table 4.7c and code is in Appendix A.

We can make several generalizations by examining Table 4.7. Since summing (SUM) and adding (±) are similar operations, the principal difference between Implementations 1 and 3 (in the packed version) is the number of startups--approximately 4 $NN_s$ versus $2N^2$ and $4NN_s$ versus $N^2$ for the operations * and ±, respectively. Hence, Implementation 1 is superior or approximately equivalent to Implementation 3 in multiplicative

| | PACKED SCHEME | | | UNPACKED SCHEME | | |
|---|---|---|---|---|---|---|
| | SU | V OP | S OP | SU | V OP | S OP |
| * | $(4N-6)N_s$ | $(2N^2-4)N_s$ | $4N_s$ | $4NN_s$ | $4N^2N_s$ | 0 |
| ± | $(4N-6)N_s$ | $(2N^2-4)N_s$ | $4N_s$ | $4NN_s$ | $4N^2N_s$ | 0 |

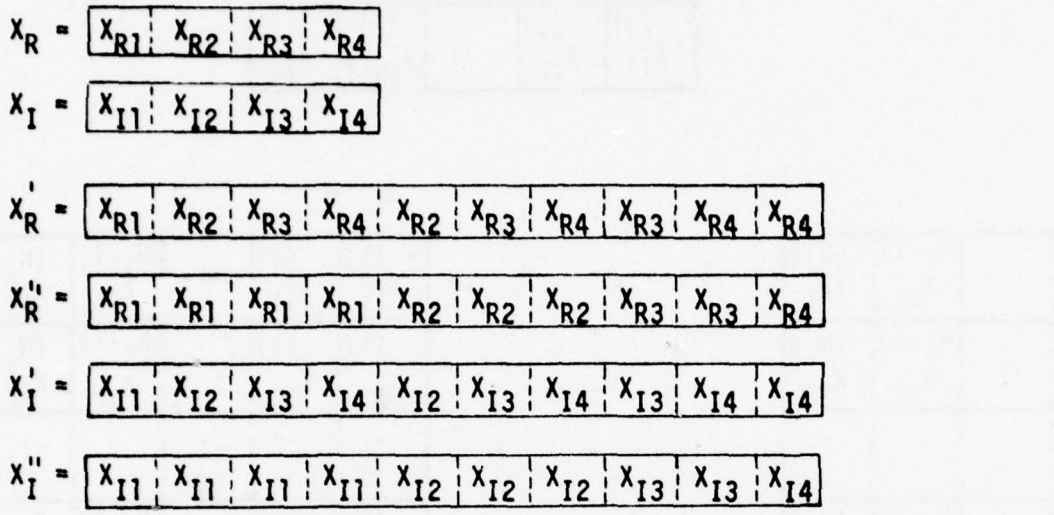
Table 4.7a Real Operation Counts as Function of Sample Size = $N_s$ for Implementation 1 for Calculating the Sample Covariance Matrix on a Vector Pipeline Processor

| | PACKED SCHEME | | | UNPACKED SCHEME | | |
|---|---|---|---|---|---|---|
| | SU | V OP | S OP | SU | V OP | S OP |
| * | $4N_s$ | $(2N^2 + 2N)N_s$ | 0 | $4N_s$ | $4N^2N_s$ | 0 |
| ± | $4N_s$ | $(2N^2 + 2N)N_s$ | 0 | $4N_s$ | $4N^2N_s$ | 0 |
| MOVE | $(4N-4)N_s$ | $(2N^2+2N-4)N_s$ | $4N_s$ | $4NN_s$ | $4N^2N_s$ | 0 |

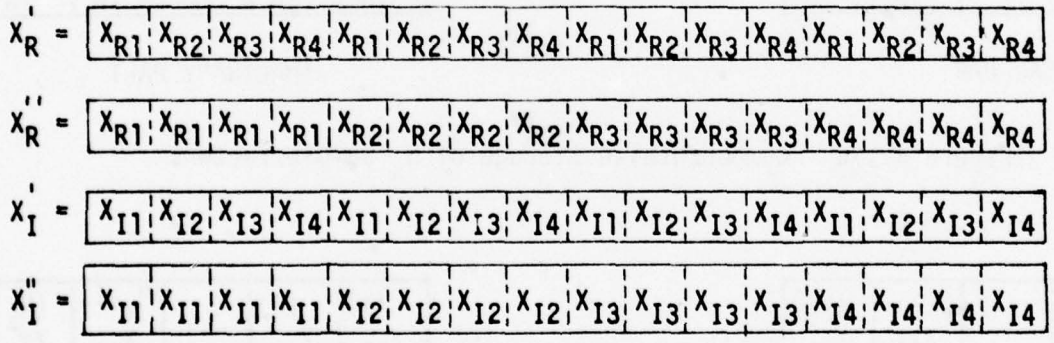
Table 4.7b Real Operation as a Function of Sample Size = $N_s$ for Implementation 2 for Calculating the Sample Covariance Matrix on a Vector Pipeline Processor

| | PACKED SCHEME | | | UNPACKED SCHEME | | |
|---|---|---|---|---|---|---|
| | SU | V OP | S OP | SU | V OP | S OP |
| * | $2N^2$ | $2N^2N_s$ | 0 | $2N^2$ | $2N^2N_s$ | 0 |
| ± | 0 | 0 | 0 | 0 | 0 | 0 |
| MOVE | 0 | 0 | 0 | 0 | 0 | $N^2-N$ |
| SUM | $N^2$ | $2N^2N_s$ | 0 | $N^2$ | $2N^2N_s$ | 0 |

Table 4.7c Real Operation Counts as a Function of Sample Size = $N_s$ for Implementation 3 for Calculating the Sample Covariance Matrix on a Vector Pipeline Processor




PACKED SCHEME




UNPACKED SCHEME

Figure 4.16 Storage Schemes of "Stretched" Input Voltage Vector $X = X_R + iX_I$ Required by Implementation 2 for a Sample Covariance Matrix Calculation When N=4

$j^{th}$ Sample Vector =

| | | | | |
|---|---|---|---|---|
| $X_{R1}^{(j)}$ | $X_{R2}^{(j)}$ | ... | $X_{RN-1}^{(j)}$ | $X_{RN}^{(j)}$ |

| | | | | |
|---|---|---|---|---|
| $X_{I1}^{(j)}$ | $X_{I2}^{(j)}$ | ... | $X_{IN-1}^{(j)}$ | $X_{IN}^{(j)}$ |

| | | | | |
|---|---|---|---|---|
| $X_{R1}^{(1)}$ | $X_{R1}^{(2)}$ | ... | $X_{R1}^{(N_s-1)}$ | $X_{R1}^{(N_s)}$ |
| $X_{R2}^{(1)}$ | $X_{R2}^{(2)}$ | ... | $X_{R2}^{(N_s-1)}$ | $X_{R2}^{(N_s)}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $X_{RN-1}^{(1)}$ | $X_{RN-1}^{(2)}$ | ... | $X_{RN-1}^{(N_s-1)}$ | $X_{RN-1}^{(N_s)}$ |
| $X_{RN}^{(1)}$ | $X_{RN}^{(2)}$ | ... | $X_{RN}^{(N_s-1)}$ | $X_{RN}^{(N_s)}$ |

REAL PART

| | | | | |
|---|---|---|---|---|
| $X_{I1}^{(1)}$ | $X_{I1}^{(2)}$ | ... | $X_{I1}^{(N_s-1)}$ | $X_{I1}^{(N_s)}$ |
| $X_{I2}^{(1)}$ | $X_{I2}^{(2)}$ | ... | $X_{I2}^{(N_s-1)}$ | $X_{I2}^{(N_s)}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $X_{IN-1}^{(1)}$ | $X_{IN-1}^{(2)}$ | ... | $X_{IN-1}^{(N_s-1)}$ | $X_{IN-1}^{(N_s)}$ |
| $X_{IN}^{(1)}$ | $X_{IN}^{(2)}$ | ... | $X_{IN}^{(N_s-1)}$ | $X_{IN}^{(N_s)}$ |

IMAGINARY PART

Figure 4.17a Componentwise Storage of $N_s$ Sample Vectors

| | | | | |
|---|---|---|---|---|
| $X_{R1}^{(1)}$ | $X_{R1}^{(2)}$ | ... | $X_{R1}^{(N_s-1)}$ | $X_{R1}^{(N_s)}$ |
| $X_{R2}^{(1)}$ | $X_{R2}^{(2)}$ | ... | $X_{R2}^{(N_s-1)}$ | $X_{R2}^{(N_s)}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $X_{RN-1}^{(1)}$ | $X_{RN-1}^{(2)}$ | ... | $X_{RN-1}^{(N_s-1)}$ | $X_{RN-1}^{(N_s)}$ |
| $X_{RN}^{(1)}$ | $X_{RN}^{(2)}$ | ... | $X_{RN}^{(N_s-1)}$ | $X_{RN}^{(N_s)}$ |

REAL PART

| | | | | |
|---|---|---|---|---|
| $X_{I1}^{(1)}$ | $X_{I1}^{(2)}$ | ... | $X_{I1}^{(N_s-1)}$ | $X_{I1}^{(N_s)}$ |
| $X_{I2}^{(1)}$ | $X_{I2}^{(2)}$ | ... | $X_{I2}^{(N_s-1)}$ | $X_{I2}^{(N_s)}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| $X_{IN-1}^{(1)}$ | $X_{IN-1}^{(2)}$ | ... | $X_{IN-1}^{(N_s-1)}$ | $X_{IN-1}^{(N_s)}$ |
| $X_{IN}^{(1)}$ | $X_{IN}^{(2)}$ | ... | $X_{IN}^{(N_s-1)}$ | $X_{IN}^{(N_s)}$ |

IMAGINARY PART

Figure 4.17b Vectorwise Storage of $N_s$ Sample Vectors

startups as long as $N_s \leq N/2$, and is superior or equivalent in additive startups whenever $N_s \leq N/4$. To compare either implementation with Implementation 2 involves a similar comparison of MOVE times with $*$ startups and $\pm$ startups, but here no generalization can be made since the comparisons depend on the values of $t_{su}$ and $t_{op}$ for these operations. These analyses must be done for each machine to be used.

A word remains to be said about fault tolerance, or what happens when the system is notified that certain components of the sample vectors are invalid. Action must be taken here, since it is particularly simple to take care of at this point, but complicated if the processing has reached the next stage, solving $MW = \overline{S}$. Deleting the $i^{th}$ component (weight) from the system is here equivalent to deleting the $i^{th}$ row and $i^{th}$ column of the sample covariance matrix. This means removing the $i^{th}$ component from the current and future sample vectors, a simple data movement, and removing the $i^{th}$ row and column of the matrix. Since the matrix is stored rowwise (in Implementations 1 through 3), deleting the $i^{th}$ row is also a simple data movement, but the $i^{th}$ column deletion is more difficult. Each row must be moved a different distance; so while one more instruction (and hence 1 startup) sufficed for deleting the row, N are needed for deleting the column. In addition, in the unpacked scheme, the column components are located in evenly spaced memory locations; thus, calculating source and destination locations is easy, but more care must be taken for the packed case.

4.3.2.3 Direct Methods of Solving MW = $\overline{S}$ on a Vector Pipeline Processor

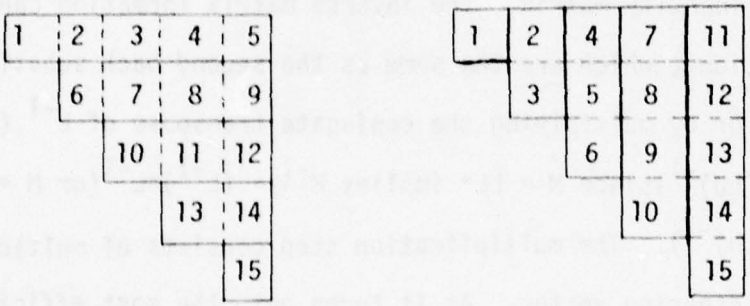
From the section on mathematical techniques, we recall that we can classify the direct methods of solving MW = $\overline{S}$ into two groups on the basis of their asymptotic operation counts: the $O(N^{\log_2 7})$ methods and the $O(N^3)$ methods. We will confine our attention to the second group for several reasons. First, as was shown above, the constant factors in the operation counts for the $O(N^{\log_2 7})$ methods are so large that the operation counts do not become lower than those of the other group until N, the number of weights, becomes very large. Second, the $O(N^{\log_2 7})$ methods depend on partitioning the matrix M, resulting in many short vector operations and large startup time. Third, since these methods are most simply implemented recursively, there is a great deal of overhead involved in subroutine calls and updating stacks, and a great deal of core needed to maintain the stacks; and since the problem we are dealing with involves a problem of dimension 200 or more, this extra overhead, and extra storage, is very important.

The basic $O(N^3)$ algorithms, whose implementations we will discuss (with their abbreviations in parentheses) are: Gauss-Jordan Elimination (GJ), Gaussian Elimination (GE), Cholesky decomposition with square roots (LL*) and Cholesky decomposition without square roots (LDL*). The last two abbreviations arise from the form of the triangular decomposition generated by the Cholesky methods, M = LL* and M = LDL*, where, in the first case, L is a lower triangular matrix with positive real diagonal elements, and in the second case, L is a lower triangular matrix with unit diagonal and D is a diagonal matrix with positive real entries. GE provides the same decomposition as LDL*. In general, GE, LDL* and LL* have different implementations, depending on the

storage schemes used for M,L* (or U), and S, as well as on the number of steering vectors S and when they are available. GJ has essentially one implementation and requires unpacked storage of M (see Figure 4.14) and availability of S as soon as M is available. Each algorithm has two kinds of implementations it can be used with: inverse matrix multiplication (IMM), where the inverse matrix is calculated and then multiplied by each steering vector; and back substitution (BS) on the steering vectors directly. IMM can be divided into three stages: decomposition with identity matrix augmentation, inverse matrix formation, and multiplication. The first stage calculates $(LD)^{-1}$ (or $L^{-1}$, for the LL* method) by adjoining the identity matrix to M, triangularizing M using either GE, LDL* or LL*, and performing the same operations on the identity matrix. The inverse matrix formation can be done by back substitutions, which are the same as the second back substitution discussed below, or by multiplying the conjugate transpose of $L^{-1}$, $(L^{-1})^*$, by $L^{-1}$ (or $(L^{-1})^*$ by $(LD)^{-1}$) since $M = LL^*$ implies $M^{-1} = (L^{-1})^* L^{-1}$ (or $M = LDL^*$ implies $M^{-1} = (L^{-1})^* (LD)^{-1}$). The multiplication step consists of multiplying the inverse matrix by the steering vector. As it turns out, the most efficient algorithm collapses the last two stages by multiplying each steering vector first by $L^{-1}$ and then by $(L^{-1})^*$ (or by $(LD)^{-1}$ and $(L^{-1})^*$). BS can also be divided into three stages: decomposition, first back substitution, and second back substitution.

Decomposition consists of calculating $L^*$ (and D, for GE and $LDL^*$); the first back substitution solves the triangular system $LT = \overline{S}$ (or $LDT = \overline{S}$), where T is a temporary variable; and the second back substitution solves $L^*W = T$. The decomposition is affected by the storage scheme for M (packed

or unpacked), and the back substitutions by the storage schemes for L (rowwise or columnwise--see Fig.4.18) and for S (vectorwise or componentwise--see Figs. 4.17a and 4.17b). If S is available as soon as M is, it may be adjoined to M to form an augmented matrix that may be used to collapse the first back substitution into the decomposition, eliminating some of the startups the first back substitution would have needed. Augmentation has four forms, depending on whether M is packed or unpacked, and whether S is adjoined vectorwise or componentwise. If S is adjoined vectorwise, it does not matter where in core it is kept with respect to M, but the storage scheme for componentwise adjoining depends on M and is shown in Figs. 4.19a and 4.19b.




Rowwise Columnwise

Figure 4.18. Storage Schemes for the Factor $L^*$ of the Matrix M

| | | | | | | |
|---|---|---|---|---|---|---|
| $M_{R11}$ | $M_{R12}$ | $M_{R13}$ | $M_{R14}$ | $S_{R1}^{(1)}$ | $S_{R1}^{(2)}$ | $S_{R1}^{(3)}$ |
| | $M_{R22}$ | $M_{R23}$ | $M_{R24}$ | $S_{R2}^{(1)}$ | $S_{R2}^{(2)}$ | $S_{R2}^{(3)}$ |
| | | $M_{R33}$ | $M_{R34}$ | $S_{R3}^{(1)}$ | $S_{R3}^{(2)}$ | $S_{R3}^{(3)}$ |
| | | | $M_{R44}$ | $S_{R4}^{(1)}$ | $S_{R4}^{(2)}$ | $S_{R4}^{(3)}$ |

REAL PART

| | | | | | | |
|---|---|---|---|---|---|---|
| $M_{I11}$ | $M_{I12}$ | $M_{I13}$ | $M_{I14}$ | $S_{I1}^{(1)}$ | $S_{I1}^{(2)}$ | $S_{I1}^{(3)}$ |
| | $M_{I22}$ | $M_{I23}$ | $M_{I24}$ | $S_{I2}^{(1)}$ | $S_{I2}^{(2)}$ | $S_{I2}^{(3)}$ |
| | | $M_{I33}$ | $M_{I34}$ | $S_{I3}^{(1)}$ | $S_{I3}^{(2)}$ | $S_{I3}^{(3)}$ |
| | | | $M_{I44}$ | $S_{I4}^{(1)}$ | $S_{I4}^{(2)}$ | $S_{I4}^{(3)}$ |

IMAGINARY PART

Figure 4.19a Packed Storage Scheme for Componentwise Augmentation of M when N = 4 and K = 3

| | | | | | | |
|---|---|---|---|---|---|---|
| $M_{R11}$ | $M_{R12}$ | $M_{R13}$ | $M_{R14}$ | $S_{R1}^{(1)}$ | $S_{R1}^{(2)}$ | $S_{R1}^{(3)}$ |
| $M_{R21}$ | $M_{R22}$ | $M_{R23}$ | $M_{R24}$ | $S_{R2}^{(1)}$ | $S_{R2}^{(2)}$ | $S_{R2}^{(3)}$ |
| $M_{R31}$ | $M_{R32}$ | $M_{R33}$ | $M_{R34}$ | $S_{R3}^{(1)}$ | $S_{R3}^{(2)}$ | $S_{R3}^{(3)}$ |
| $M_{R41}$ | $M_{R42}$ | $M_{R43}$ | $M_{R44}$ | $S_{R4}^{(1)}$ | $S_{R4}^{(2)}$ | $S_{R4}^{(3)}$ |

REAL PART

| | | | | | | |
|---|---|---|---|---|---|---|
| $M_{I11}$ | $M_{I12}$ | $M_{I13}$ | $M_{I14}$ | $S_{I1}^{(1)}$ | $S_{I1}^{(2)}$ | $S_{I1}^{(3)}$ |
| $M_{I21}$ | $M_{I22}$ | $M_{I23}$ | $M_{I24}$ | $S_{I2}^{(1)}$ | $S_{I2}^{(2)}$ | $S_{I2}^{(3)}$ |
| $M_{I31}$ | $M_{I32}$ | $M_{I33}$ | $M_{I34}$ | $S_{I3}^{(1)}$ | $S_{I3}^{(2)}$ | $S_{I3}^{(3)}$ |
| $M_{I41}$ | $M_{I42}$ | $M_{I43}$ | $M_{I44}$ | $S_{I4}^{(1)}$ | $S_{I4}^{(2)}$ | $S_{I4}^{(3)}$ |

IMAGINARY PART

Figure 4.19b Unpacked Storage Scheme for Componentwise Augmentation of M when N = 4 and K = 3

The implementations of these algorithms and their variations are in Appendix B and their corresponding operation counts in Table 4.8. The operations counts are functions of the number of weights, N, and the number of steering vectors, K. GJ has only one version and is shown in Table 4.8a. The different decompositions are shown in Tables 4.8b and 4.8c; decompositions with an augmented identity matrix, in 4.8d. The first back substitution is in 4.8e, and the second back substitution, in Table 4.8f. $LDL^*$ and GE are shown together in the back substitutions since the only difference in the algorithm is in the decomposition.

Whether M is packed or not does not affect the operation count, and so no distinction is made in the tables. It should be noted, however, that only Cholesky ($LDL^*$ and $LL^*$) has packed implementations. The abbreviations for the different storage schemes are given in Fig. 4.20. The back substitution algorithms assume no spatial relation of the sample covariance matrix M and the steering vectors S, so if the storage of the two arrays is overlapped (as in the componentwise augmented case), the subscript calculation will have to be redone but this will not affect the operation count since here, as in all other tables, only floating-point operations are counted.

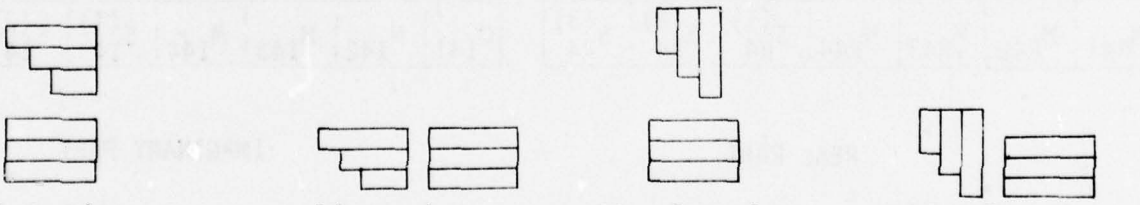



Figure 4.20 Abbreviations for Different Storage Schemes Used in Table 4.8 and Appendix B

Table 4.8a Real Operation Counts for GJ

| | SU | V OP | S OP |
|---|---|---|---|
| * | $4N^2-2N$ | $2N^3+N^2(4K-3)+N(1-2K)$ | 0 |
| $\pm$ | $2N^2-4N$ | $2N^3+N^2(4K-4)+N(2-4K)$ | 0 |
| / | 0 | 0 | N |

Table 4.8b Real Operation Counts for Decomposition

GE

| | SU | V OP | S OP |
|---|---|---|---|
| * | $2N^2-8$ | $\frac{4}{3}N^3-N^2-\frac{1}{3}N-6$ | 4 |
| $\pm$ | $2N^2-2N-4$ | $\frac{4}{3}N^3-2N^2+\frac{2}{3}N-4$ | 2 |
| / | 0 | 0 | N |

LDL*

| | SU | V OP | S OP |
|---|---|---|---|
| * | $2N^2-4N$ | $\frac{2}{3}N^3-\frac{14}{3}N+4$ | 2N |
| $\pm$ | $2N^2-6N+4$ | $\frac{2}{3}N^3-N^2-\frac{11}{3}N+6$ | 2N-2 |
| / | 0 | 0 | N |
| MOVE | 2N-4 | $N^2-N-2$ | 2 |

LL*

| | SU | V OP | S OP |
|---|---|---|---|
| * | $2N^2-4N$ | $\frac{2}{3}N^3-\frac{14}{3}N+4$ | 2N |
| $\pm$ | $2N^2-6N+4$ | $\frac{2}{3}N^3-N^2-\frac{11}{3}N+6$ | 2N-2 |
| / | 0 | 0 | N |
| $\sqrt{\ }$ | 0 | 0 | N |

Table 4.8c Real Operation Counts for Decomposition with Augmentation

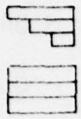
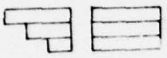
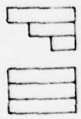
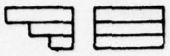
| | | SU | V OP | S OP |
|---|---|---|---|---|
| GE | | | | |
|  | * | $2N^2+N(4K)-8-6K$ | $\frac{4}{3}N^3+N^2(2K-1)-\frac{1}{3}N-6-4K$ | $4K+4$ |
|  | $\pm$ | $2N^2+N(4K-2)-4-8K$ | $\frac{4}{3}N^3+N^2(2K-2)+N(\frac{2}{3}-2K)-4-4K$ | $4K+2$ |
| | / | 0 | 0 | N |
| GE | | | | |
|  | * | $2N^2$ | $\frac{4}{3}N^3+N^2(2K-1)-\frac{1}{3}N$ | 0 |
| | $\pm$ | $2N^2-2N$ | $\frac{4}{3}N^3+N^2(2K-2)+N(\frac{2}{3}-2K)$ | 0 |
| | / | 0 | 0 | N |
| LDL* | | | | |
|  | * | $2N^2+N(4K-4)-6K$ | $\frac{2}{3}N^3+2KN^2-\frac{14}{3}N-4K+4$ | $2N+4K$ |
|  | $\pm$ | $2N^2+N(4K-6)+4-8K$ | $\frac{2}{3}N^3+N^2(2K-1)+N(-\frac{11}{3}-2K)-4K+6$ | $2N+4K-2$ |
| | / | 0 | 0 | N |
| | MOVE | $2N-4$ | $N^2-N-2$ | 2 |
| LDL* | * | $2N^2$ | $\frac{2}{3}N^3+2KN^2-\frac{2}{3}N$ | 0 |
| | $\pm$ | $2N^2-2N$ | $\frac{2}{3}N^3+N^2(2K-1)+N(\frac{1}{3}-2K)$ | 0 |
| | / | 0 | 0 | N |
| | MOVE | $2N-2$ | $N^2+N(2K-1)-2K$ | 0 |
| LL* | | | | |
| | * | $2N^2+N(4K-4)-8K$ | $\frac{2}{3}N^3+2KN^2+N(-\frac{14}{3}-2K)-4K+4$ | $N(2K+2)+4K$ |
| | $\pm$ | $2N^2+N(4K-6)+4-8K$ | $\frac{2}{3}N^3+N^2(2K-1)+N(-\frac{11}{3}-2K)-4K+6$ | $2N+4K-2$ |
| | / | 0 | 0 | N |
| | $\sqrt{}$ | 0 | 0 | N |

Table 4.8c Real Operation Counts for Decomposition with Augmentation (Cont'd)

LL*



| | | | |
|---|---|---|---|
| * | $2N^2$ | $\frac{2}{3}N^3 + 2KN^2 - \frac{2}{3}N$ | 0 |
| $\pm$ | $2N^2-2N$ | $\frac{2}{3}N^3 + N^2(2K-1) + N(\frac{1}{3} - 2K)$ | 0 |
| / | 0 | 0 | N |
| $\sqrt{\ }$ | 0 | 0 | N |

Table 4.8d Real Operation Counts for Decomposition with Identity Matrix Augmentation

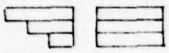
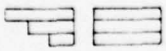
| | | SU | V OP | S OP |
|---|---|---|---|---|
| GE | | | | |
| | * | $4N^2 - 8N + 1$ | $2N^3 - 2N^2 - 3N+2$ | 4N+1 |
| | $\pm$ | $4N^2 - 12N + 8$ | $2N^3 - 4N^2 - 2N + 4$ | 4N-2 |
| | / | 0 | 0 | N |
| | MOVE | 2N-4 | $N^2-N-2$ | 1 |
| GE | | | | |
|  | * | $2N^2$ | $2N^3+2N^2-2N$ | 0 |
| | $\pm$ | $2N^2-2N$ | $2N^3-2N$ | 0 |
| | / | 0 | 0 | N |
| LDL* | | | | |
| | * | $4N^2 - 12N + 9$ | $\frac{4}{3} N^3 - N^2 - \frac{22}{3} N + 12$ | 6N-3 |
| | $\pm$ | $4N^2-16N + 16$ | $\frac{4}{3} N^3 - 3N^2 - \frac{19}{3} N + 14$ | 6N-6 |
| | / | 0 | 0 | N |
| | MOVE | 4N-8 | $2N^2 - 2N - 4$ | 3 |
| LDL* | | | | |
| | * | $2N^2$ | $\frac{4}{3} N^3 + N^2 - \frac{4}{3} N$ | 0 |
|  | $\pm$ | $2N^2 - 2N$ | $\frac{4}{3} N^3 - N^2 - \frac{1}{3} N$ | 0 |
| | / | 0 | 0 | N |
| | MOVE | 2N-2 | $2N^2 - 3 N+1$ | 0 |

Table 4.8d Real Operation Counts for Decomposition with Identity Matrix Augmentation (Cont'd)

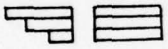
| LL* | | SU | V OP | S OP |
|---|---|---|---|---|
| | * | $4N^2 - 13N + 10$ | $\frac{4}{3}N^3 - \frac{3}{2}N^2 - \frac{47}{6}N + 11$ | $N^2 + 5N - 2$ |
| | $\pm$ | $4N^2 - 16N + 16$ | $\frac{4}{3}N^3 - 3N^2 - \frac{19}{3}N + 14$ | $6N - 6$ |
| | / | 0 | 0 | N |
| | $\sqrt{\ }$ | 0 | 0 | N |
| | MOVE | $2N - 4$ | $N^2 - N - 2$ | 1 |
| LL* | | | | |
| | * | $2N^2$ | $\frac{4}{3}N^3 + N^2 - \frac{4}{3}N$ | 0 |
|  | $\pm$ | $2N^2 - 2N$ | $\frac{4}{3}N^3 - N^2 - \frac{1}{3}N$ | 0 |
| | / | 0 | 0 | N |
| | $\sqrt{\ }$ | 0 | 0 | N |

Table 4.8e Real Operation Counts for First Back Substitutions

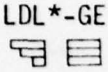
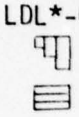
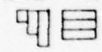
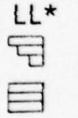
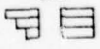
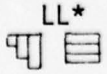
| | | SU | V OP | S OP |
|---|---|---|---|---|
| LDL*-GE | * | $N(4K)-6K$ | $N^2(2K)-4K$ | $4K$ |
| | $\pm$ | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $4K$ |
| LDL*-GE  | * | $2N^2$ | $N^2(2K)$ | 0 |
| | $\pm$ | $2N^2-2N$ | $N^2(2K)-N(2K)$ | 0 |
| LDL*-GE  | * | $N(4K)-6K$ | $N^2(2K)-4K$ | $4K$ |
| | $\pm$ | 0 | 0 | $N(2K)$ |
| | SUM | $N(2K)-4K$ | $N^2(2K)+N(-2K)-4K$ | 0 |
| LDL*-GE  | * | Same as LDL*-GE  above | | |
| LL*  | * | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $N(2K)+4K$ |
| | $\pm$ | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $4K$ |
| LL*  | * | $2N^2$ | $N^2(2K)$ | 0 |
| | $\pm$ | $2N^2-2N$ | $N^2(2K)-N(2K)$ | 0 |
| LL* | * | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $N(2K)+4K$ |
| | $\pm$ | 0 | 0 | $N(2K)$ |
| | SUM | $N(2K)-4K$ | $N^2(2K)+N(-2K)-4K$ | 0 |
| LL*  | * | Same as LL*  above | | |
| | $\pm$ | | | |

Table 4.8f Real Operation Counts for Second Back Substitutions

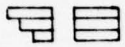
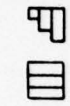
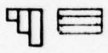
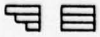
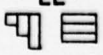
| | | SU | V OP | S OP |
|---|---|---|---|---|
| LDL*-GE | * | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $4K$ |
| | ± | $0$ | $0$ | $N(2K)$ |
| | SUM | $N(2K)-4K$ | $N^2(2K)+N(-2K)-4K$ | $0$ |
| LDL*-GE  | * | $2N^2-2N$ | $N^2(2K)+N(-2K)$ | $0$ |
| | ± | $2N^2-2N$ | $N^2(2K)+N(-2K)$ | $0$ |
| LDL*-GE  | * | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $4K$ |
| | ± | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $4K$ |
| LDL*-GE  | * | Same as LDL*-GE above | | |
| | ± | | | |
| LL* | * | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $N(2K)+4K$ |
| | ± | $0$ | $0$ | $N(2K)$ |
| | SUM | $N(2K)-4K$ | $N^2(2K)+N(-2K)-4K$ | $0$ |
| LL*  | * | $2N^2$ | $N^2(2K)$ | $0$ |
| | ± | $2N^2-2N$ | $N^2(2K)+N(-2K)$ | $0$ |
| LL* | * | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $N(2K)+4K$ |
| | ± | $N(4K)-8K$ | $N^2(2K)+N(-2K)-4K$ | $4K$ |
| LL*  | * | Same as LL* above | | |
| | ± | | | |

Table 4.9 Dominating Terms of Real Operation Counts; $\binom{m}{n}$ Denotes "either m or n" Depending on Algorithm, SUM Counts are Included in $\pm$ Counts

| | | SU | V OP |
|---|---|---|---|
| GJ | | | |
| | * | $4N^2$ | $2N^3$ |
| | $\pm$ | $2N^2$ | $2N^3$ |
| Decompositions only | | | |
| GE | | | |
| | * | $2N^2$ | $\frac{4}{3}N^3$ |
| | $\pm$ | $2N^2$ | $\frac{4}{3}N^3$ |
| LDL* | | | |
| | * | $2N^2$ | $\frac{2}{3}N^3$ |
| | $\pm$ | $2N^2$ | $\frac{2}{3}N^3$ |
| LL* | | | |
| | * | $2N^2$ | $\frac{2}{3}N^3$ |
| | $\pm$ | $2N^2$ | $\frac{2}{3}N^3$ |
| Decompositions with Augmentation | | | |
| GE | | | |
| | * | $2N^2$ | $\frac{4}{3}N^3$ |
| | $\pm$ | $2N^2$ | $\frac{4}{3}N^3$ |
| LDL* | | | |
| | * | $2N^2$ | $\frac{2}{3}N^3$ |
| | $\pm$ | $2N^2$ | $\frac{2}{3}N^3$ |
| LL* | | | |
| | * | $2N^2$ | $\frac{2}{3}N^3$ |
| | $\pm$ | $2N^2$ | $\frac{2}{3}N^3$ |

Table 4.9 (Cont'd)

Decomposition with Identity Matrix Augmentation

| | | SU | V OP |
|---|---|---|---|
| GE | | | |
| | * | $\binom{4}{2}N^2$ | $2N^3$ |
| | $\pm$ | $\binom{4}{2}N^2$ | $2N^3$ |
| LDL* | | | |
| | * | $\binom{4}{2}N^2$ | $\frac{4}{3}N^3$ |
| | $\pm$ | $\binom{4}{2}N^2$ | $\frac{4}{3}N^3$ |
| LL* | | | |
| | * | $\binom{4}{2}N^2$ | $\frac{4}{3}N^3$ |
| | $\pm$ | $\binom{4}{2}N^2$ | $\frac{4}{3}N^3$ |
| First Back Substitution | | | |
| LDL* or GE | | | |
| | * | $\binom{4K}{2N}N$ | $2KN^2$ |
| | $\pm$ | $\binom{4K}{2N}N$ | $2KN^2$ |
| LL* | | | |
| | * | $\binom{4K}{2N}N$ | $2KN^2$ |
| | $\pm$ | $\binom{4K}{2N}N$ | $2KN^2$ |
| Second Back Substitution | | | |
| LDL* or GE | | | |
| | * | $\binom{4K}{2N}N$ | $2KN^2$ |
| | $\pm$ | $\binom{4K}{2N}N$ | $2KN^2$ |
| LL* | | | |
| | * | $\binom{4K}{2N}N$ | $2KN^2$ |
| | $\pm$ | $\binom{4K}{2N}N$ | $2KN^2$ |

First, we will make some general comments about the operation counts, whose dominating terms are summarized in Table 4.9. For the decompositions, both augmented and non-augmented, we note that the number of startups (SU) is of a lower order than the number of vector operations (V OP), $O(N^2)$ versus $O(N^3)$. This means that, asymptotically, SU is neglible in comparison to V OP, and that the conclusions of the algorithm analysis for sequential machines is valid here: Cholesky is best, followed by Gaussian Elimination and Gauss-Jordan. We can also show that, asymptotically, inverse matrix multiplication is worse than two back substitutions for any number of steering vectors. The decomposition stage of IMM requires $\frac{4}{3} N^3$ V OP's using Cholesky. The counts for the other stages vary considerably, depending on the form of algorithm--back substitutions and steering vector multiplication require $2N^3 + 4KN^2$ operations; multiplying $(L^{-1})^*$ by $L^{-1}$ and steering vector multiplication require $\frac{2}{3} N^3 + 4KN^2$; and multiplying the steering vectors first by $L^{-1}$ and then $(L^{-1})^*$ requires $4KN^2$, the least of all. Hence, IMM requires $\frac{4}{3} N^3 + 4KN^2$ operations in contrast to BS, which requires only $\frac{2}{3} N^3 + 4KN^2$. In addition, IMM requires at least $N^2 + N$ complex locations in contrast to $(N^2 + N)/2$ for BS, and appears to have no advantage in the number of SU's required. Hence, we will restrict our attention to $LDL^*$ and $LL^*$, augmented or not, followed by back substitutions.

The only difference between $LDL^*$ and $LL^*$ decomposition is that $LL^*$ requires N square roots, while $LDL^*$ requires data movement of approximately $N^2$ words in 2N operations. On some machines (CDC STAR, for example), the MOVE operation is comparable in time to the $\pm$ operation, and on many machines,

there is a hardware floating-point square root instruction, so $LL^*$ may well be superior to $LDL^*$, but this decision, as many others, requires a detailed knowledge of a particular processor. In fact, when one examines the implementations in Appendix B, several machine-dependent changes come to mind. The vector lengths vary from 2 up to N, but if a machine's tradeoff point for scalar loop versus vector operation is greater than 2, this minimum vector length would have to be changed. In the inner loop, where 2 multiplications and 2 additions occur in the same line, chaining would greatly speed up processing. The two adjacent calculations there, for MR and MI, are independent, and could be done in parallel. These comments also apply to the augmented versions of these implementations.

There are two basic augmentation implementations: vectorwise and componentwise. They require approximately the same number of operations, but componentwise augmentation has an advantage in the number of startups being 4KN less than its vectorwise counterpart, both in *SU and ±SU. Vectorwise augmentation is really no different from a decomposition followed by a back substitution with rowwise stored steering vectors, but componentwise augmentation takes advantage of the early data availability to arrange the data to avoid startups. It also avoids short vector and scalar operations, the shortest vector operation being of length K, the number of steering vectors. However, this arrangement, interweaving the rows of the matrix with the corresponding components of the steering vectors, has its own problems. It requires storing the steering vectors in non-consecutive locations in core; and if the packed

matrix storage scheme is used, in unevenly spaced locations in core. The N real parts of the first row of the matrix would be followed by the K real parts of the first components of all the steering vectors, followed by the N-1 real parts of the second row of the matrix, and so on. This arrangement makes it difficult to store the steering vectors in their proper locations initially, and hard to access the weights later, but the magnitude of these problems is system-dependent.

Both the first and second back substitutions require approximately $2KN^2$ operations, and either $4KN$ or $2N^2$ startups, depending on the storage scheme. The problem here is that what might be the optimal storage scheme for the decomposition might not be the optimal scheme for the first back substitution, which in turn might not be best for the second one. All storage schemes require similar numbers of multiplications or additions, but some require the SUM operation, some $4KN$ startups, and some $2N^2$ startups. The tradeoff point for startups occurs approximately when $4KN = 2N^2$, or $K = N/2$. When K exceeds N/2, the $2N^2$ algorithm is tentatively recommended, and otherwise the $4KN$ method is recommended. Methods requiring the transpose of the matrix have been included, even though they are outside the scope of our assumptions, since they provide optimal performance in the second back substitution when $K < N/2$. Under our assumptions, though, if $K < N/2$, componentwise storage of the steering vectors is recommended, since this is optimal for both back substitutions. Other combinations require individual analysis with respect to a particular computer.

### 4.3.3 Implementations of Algorithms for Determining Adaptive Weights on Vector Pipeline Processors

#### 4.3.3.1 CDC STAR-100

The CDC STAR-100, or STAR, for short, is a large-scale, high-speed, multipurpose vector pipeline computer. Its basic configuration is shown in Figure 4.21. (The source of this summary and these figures is the CDC STAR-100 Hardware Reference Manual, 1975.)

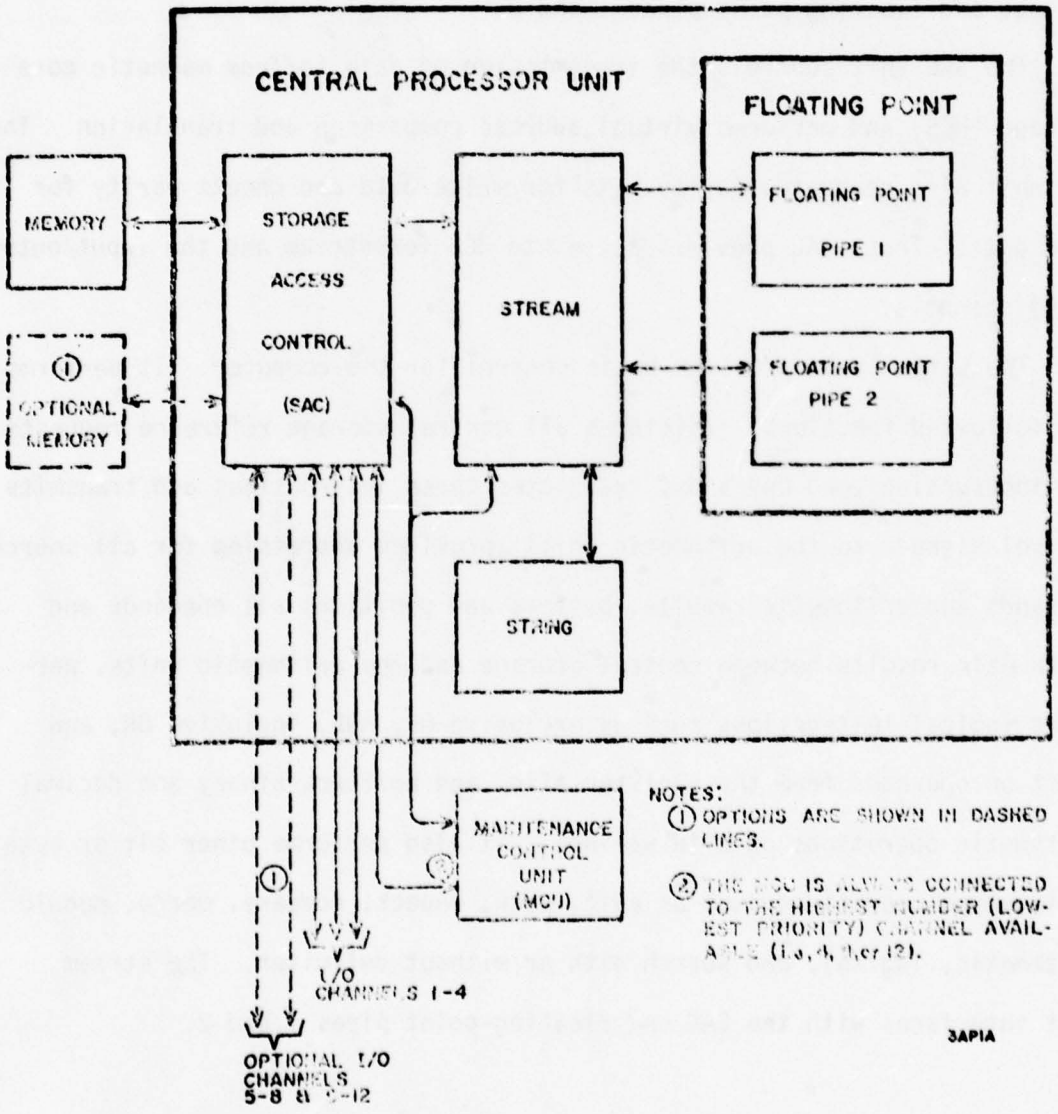



Figure 4.21 Basic CDC STAR-100 Configuration

Magnetic core storage (MCS) consists of 524,288 66-bit words (64 data bits and 2 parity bits), physically arranged as 65,536 528-bit words. MCS is addressable to the bit level.

The central processor unit (CPU) in Figure 4.21 consists of the following functional areas: storage access control (SAC), stream and string, and floating-point pipes 1 and 2.

The SAC unit controls the transmission of data to/from magnetic core storage (MCS) and performs virtual address comparison and translation. The SAC unit also generates parity bits for write data and checks parity for read data. Thus, SAC provides access to MCS for stream and the input/output (I/O) channels.

The stream unit provides basic control for the computer. It performs the following functions: initiates all central storage reference requests for instructions and operands, translates these instructions and transmits control signals to the arithmetic units, provides addressing for all source operands and arithmetic results, buffers and positions all operands and arithmetic results between central storage and the arithmetic units, performs logical instructions such as exclusive OR, AND, inclusive OR, and shift on operands from the register file, and performs binary and decimal arithmetic operations on byte strings. It also performs other bit or byte string-type operations such as edit, pack, unpack, compare, merge, modulo arithmetic, logical, and search with or without delimiter. The stream unit interfaces with the SAC and floating-point pipes 1 and 2.

Floating-point numbers in the computer are two lengths, 32 bits and 64 bits. The 32-bit format has an 8-bit exponent and a 24-bit coefficient (Figure 4.22). The 64-bit format has a 16-bit exponent and a 48-bit coefficient. The leftmost bit of each exponent and coefficient is the sign bit.

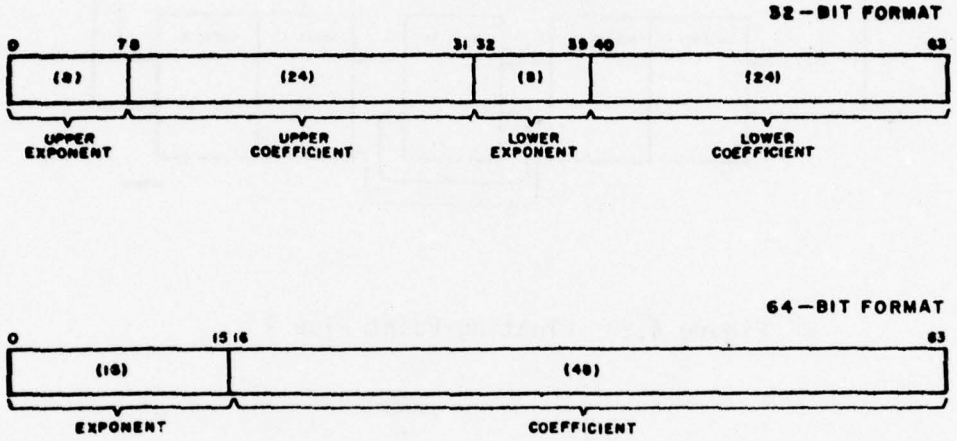



Figure 4.22 . Operand Formats

The floating-point arithmetic hardware is divided into two units or pipes. Pipe 1 (Figure 4.23) performs register add, register subtract, register multiply, and all vector arithmetic instructions except divide and square root. Pipe 2 (Figure 4.24) performs register divide, register square root, and all vector instructions. This organization of hardware allows optimum performance for both register and vector divide operations. For vector operations common to both pipe 1 and pipe 2, the data are divided in half with every second pair of 64-bit operands going to pipe 2 (that is, first pair, third pair, etc.) and every second pair (that is, second pair, fourth pair, etc.) to pipe 1. In the 32-bit model, each pipe divides in half to become two 32-bit pipes. Therefore, two pair of operands go alternatively to each pipe.

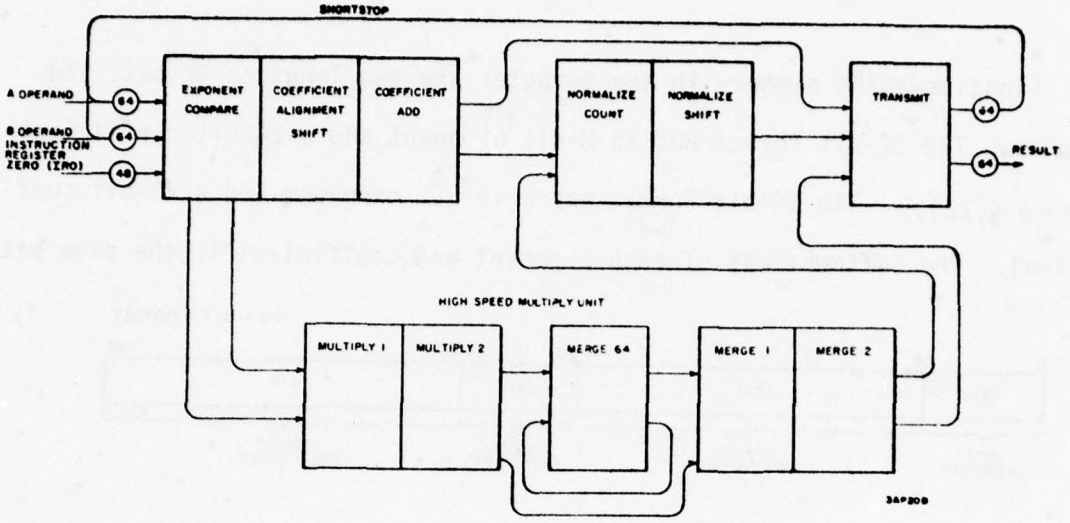



Figure 4.23 Floating-Point Pipe 1

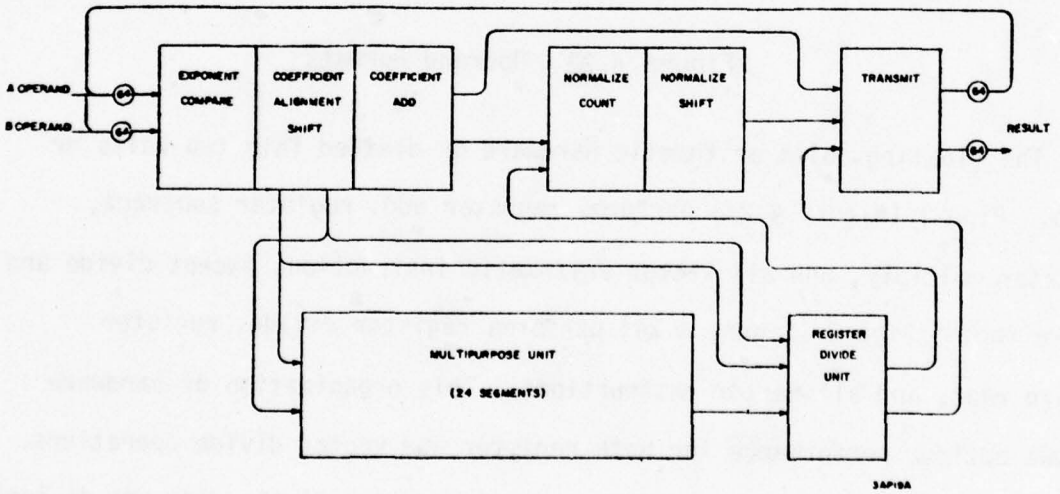



Figure 4.24 Floating-Point Pipe 2

Floating-point pipe 1 receives operands from the stream unit, performs the instructed operation, and returns the results to the stream unit. Pipe 1 performs arithmetic operations on operands in floating-point format and address operations on nonfloating-point numbers. Arithmetic operations include such operations as add, subtract, multiply, truncate, adjust exponent, contract, extend, and compare. Address-type operations are those which manipulate various parts of instructions and registers for addressing and indexing purposes. Refer to Figure 4.23 in reading the following description of some basic operations of pipe 1.

For addition and subtraction operations, the input exponents are compared in the exponent compare circuit. The difference in the two exponents is used as a shift count. This shift count determines the amount the coefficient with the smaller exponent is right shifted in the coefficient alignment section. The coefficients are added in the add section. If the operation being performed specifies normalization, the result of the add operation is fed to the normalize count. This circuit produces a shift count which controls the normalize shift network and modifies the result exponent. The transmit circuit returns the shifted result to the stream unit. If normalization is not specified, the result of the add operation is the desired result and is transmitted to stream.

If the instruction is a multiply, the operands are multiplied in the high-speed multiply unit. The result of the multiply is either returned directly to the transmit section or to the normalize count logic for normalization. The normalize count functions only for the multiply-significant instructions.

Any result from pipe 1 may be returned directly to either of the inputs of pipe 1 if the result is needed as an input operand. This process is called shortstopping and eliminates the time necessary to store the result in the register file and then retrieve it. Floating-point pipe 2 (Figure 4.24) receives operands from the stream unit, performs the instructed operation, and returns the results to the stream unit. Pipe 1 performs arithmetic operations on operands in floating-point format and address operations on nonfloating-point numbers. Arithmetic operations include such operations as add, subtract, multiply, divide, truncate, adjust exponent, contract, extend, and compare. Pipe 2 performs only two address-type operations. These are the vector add and subtract address instructions. Pipe 1 and pipe 2 are similar, except pipe 2 has a high-speed register divide unit and a multipurpose unit.

The register divide unit performs all register divide operations and binary-to-binary coded decimal (BCD) and BCD-to-binary conversions. This is a single-segment unit,and the operands loop within the unit until the result is reached.

The multipurpose unit performs the square root, vector divide, and vector multiply instructions. The multipurpose unit contains 24 segments. Each segment performs an add-type operation. The segments are arranged in four groups of six segments per group. In the 64-bit mode, the operands loop on each group, going through each group twice. In the 32-bit mode, the operands proceed from segment to segment, going through all of them only once. The multipurpose unit delivers its results to the normalize or transmit portions of pipe 2.

The STAR is currently being improved, and two new versions have been planned. The STAR-100A will be available in 1978 and the STAR-100C will be available several years later. Projected performance improvements, along with current STAR performance, are shown in Table 4.10.

We chose Implementation 1 for computing the sample covariance matrix. Implementation 2 was not chosen because it requires as many MOVE operations as Implementation 1 requires ± operations; and since the two operations are comparable in timing, the only advantage Implementation 2 would have would be * startups, a first-order effect in contrast to the second-order disadvantage of twice as many MOVE and ± operations. Implementation 3 required too much storage and too many startups.

We also chose to implement the packed, unaugmented form of LL* with one steering vector stored (necessarily) vectorwise. LL* was chosen over LDL* since the MOVE instruction is comparable in speed to the $\frac{+}{-}$ instruction, and because the square root is implemented in hardware. The packed version was chosen because of core limitations; and the unaugmented form was chosen as the most general case, with the fewest assumptions about data availability. One steering vector was used because of time and core considerations: componentwise storage will not become better than vectorwise storage until K exceeds N/2, and in vectorwise storage each steering vector is operated on independent of the others, so processing more than one steering vector will not reveal any more than just processing one. Processing N/2 steering vectors would require a great deal of core and computer time.

A great deal of time was spent in optimizing the code, which was written in FORTRAN. In performing this optimization, the distinction between

Table 4.10 64-Bit Performance Improvements of the CDC STAR 100-A over the CDC STAR-100

| | STAR 100 | STAR 100-A | RATIO STAR 100/STAR 100-A |
|---|---|---|---|
| LOAD | | | |
| A. NO CONFLICT | 1240 NSEC | 200 NSEC | 6.2* |
| B. BANK CONFLICT | 2520 | 253.3 | 9.9* |
| ADD (REGISTER) (S STOP) | 240 | 66.6 | 3.6* |
| ADD (VECTOR) | 20 | 20 | 1.0* |
| MULTIPLY (REGISTER) (S STOP) | 400 | 66.6 | 6.0* |
| MULTIPLY (VECTOR) | 40 | 40 | 1.0* |
| ISSUE | 80 | 13.3 | 6.0* |
| BRANCH | | | |
| A. IN-STACK (BX) | 760 | 120 | 6.3* |
| B. OUT-OF-STACK (BX) | 1480 | 320 | 4.6* |
| VECTOR STARTUP | 2800 | 2800 | 1+ * |

* = TIMES

(APPARENT STARTUP WILL BE LESS DUE TO PARALLEL SCALAR EXECUTION)

implementation and algorithm became clear: algorithms, discussed in the last section, are as independent of individual system architecture as possible. An implementation is wed to its machine and system configuration, and is totally dependent on whatever small quirks the machine or system may possess. A grasp of implementation problems and a logical approach to optimizing the implementation is as necessary to a successful system as starting out with the right basic algorithm. Implementations concern overhead: the language the program is written in, the best way to calculate subscripts and indices (by table look-up or calculation), what operations can be collapsed together, what values should be saved in special locations to avoid frequent indirect or indexed addressing in order to access them, what special functions can be used, and similar considerations. This analysis would have to be repeated for each system, but we feel the time spent would be justified, because, in a system where milliseconds count, it is necessary to be aware of and have expertise in the problems at all levels of the system, especially the bottom-line consisting of a few instructions the whole system depends on. Ultimately, the optimal implementation should probably be done in assembly language, but we chose FORTRAN since we thought it would give us a good feel for the problems of implementation without requiring a tremendous commitment of time and money to get timing and accuracy results, that would be accurate enough to assess system performance. Using FORTRAN provides an upper bound on the time required.

We optimized the code to the limits of our knowledge of the workings of the FORTRAN compiler, but there were many questions raised, in the course

Table 4.11 CDC STAR-100 CPU Timings in Milliseconds

| N = # Weights | Setup | Decomposition | First Back Substitution | Second Back Substitution | Zero Out Sample Co-Variance Matrix | Update Sample Covariance Matrix | Move Sample Covariance Matrix | TOTAL |
|---|---|---|---|---|---|---|---|---|
| 5 | Ø.135 | Ø.539 | Ø.292 | Ø.3Ø5 | Ø.Ø69 | Ø.313 | Ø.Ø79 | 4.549 |
| 6 | Ø.143 | Ø.776 | Ø.352 | Ø.373 | Ø.Ø69 | Ø.356 | Ø.Ø78 | 6.Ø63 |
| 7 | Ø.157 | 1.Ø66 | Ø.411 | Ø.438 | Ø.Ø69 | Ø.4Ø1 | Ø.Ø78 | 7.833 |
| 8 | Ø.168 | 1.411 | Ø.469 | Ø.511 | Ø.Ø7Ø | Ø.446 | Ø.Ø78 | 9.843 |
| 9 | Ø.171 | 1.812 | Ø.531 | Ø.589 | Ø.Ø7Ø | Ø.493 | Ø.Ø79 | 12.117 |
| 1Ø | Ø.189 | 2.269 | Ø.595 | Ø.654 | Ø.Ø7Ø | Ø.54Ø | Ø.Ø81 | 14.658 |
| 11 | Ø.192 | 2.761 | Ø.653 | Ø.721 | Ø.Ø71 | Ø.587 | Ø.Ø85 | 17.397 |
| 12 | Ø.2Ø3 | 3.318 | Ø.714 | Ø.793 | Ø.Ø71 | Ø.637 | Ø.Ø83 | 2Ø.47Ø |
| 13 | Ø.2Ø8 | 3.929 | Ø.779 | Ø.868 | Ø.Ø77 | Ø.685 | Ø.Ø81 | 23.752 |
| 14 | Ø.222 | 4.61Ø | Ø.843 | Ø.943 | Ø.Ø72 | Ø.732 | Ø.Ø81 | 27.267 |
| 15 | Ø.232 | 5.358 | Ø.91Ø | 1.Ø18 | Ø.Ø73 | Ø.779 | Ø.Ø82 | 31.Ø43 |
| 16 | Ø.241 | 6.16Ø | Ø.97Ø | 1.Ø91 | Ø.Ø73 | Ø.835 | Ø.Ø82 | 35.337 |
| 17 | Ø.25Ø | 6.97Ø | 1.Ø36 | 1.166 | Ø.Ø74 | Ø.881 | Ø.Ø83 | 39.533 |
| 18 | Ø.254 | 7.851 | 1.Ø97 | 1.247 | Ø.Ø75 | Ø.928 | Ø.Ø84 | 44.Ø16 |
| 19 | Ø.269 | 8.766 | 1.159 | 1.319 | Ø.Ø75 | Ø.978 | Ø.Ø85 | 48.837 |
| 2Ø | Ø.279 | 9.757 | 1.223 | 1.393 | Ø.Ø77 | 1.Ø28 | Ø.Ø86 | 53.94Ø |
| 21 | Ø.286 | 1Ø.841 | 1.3Ø9 | 1.486 | Ø.Ø77 | 1.Ø8Ø | Ø.Ø86 | 59.445 |
| 22 | Ø.293 | 11.947 | 1.368 | 1.563 | Ø.Ø78 | 1.129 | Ø.Ø87 | 65.Ø17 |
| 23 | Ø.3Ø2 | 13.Ø93 | 1.431 | 1.644 | Ø.Ø79 | 1.184 | Ø.Ø9Ø | 71.1Ø3 |
| 24 | Ø.311 | 14.288 | 1.497 | 1.722 | Ø.Ø84 | 1.234 | Ø.Ø91 | 77.225 |
| 25 | Ø.32Ø | 15.552 | 1.562 | 1.8Ø4 | Ø.Ø81 | 1.283 | Ø.Ø9Ø | 83.559 |
| 26 | Ø.33Ø | 16.898 | 1.622 | 1.887 | Ø.Ø82 | 1.335 | Ø.Ø91 | 9Ø.33Ø |
| 27 | Ø.339 | 18.334 | 1.693 | 1.966 | Ø.Ø85 | 1.389 | Ø.Ø98 | 97.521 |
| 28 | Ø.349 | 19.814 | 1.758 | 2.Ø55 | Ø.Ø88 | 1.446 | Ø.1ØØ | 1Ø5.14Ø |
| 29 | Ø.357 | 21.297 | 1.837 | 2.143 | Ø.Ø89 | 1.499 | Ø.1Ø3 | 112.768 |
| 3Ø | Ø.367 | 22.833 | 1.9Ø3 | 2.227 | Ø.Ø91 | 1.55Ø | Ø.1Ø5 | 12Ø.526 |
| 5Ø | Ø.94Ø | 66.523 | 3.295 | 4.Ø35 | Ø.135 | 2.665 | Ø.17Ø | 341.656 |
| 75 | Ø.782 | 156.45Ø | 5.2Ø5 | 6.Ø93 | Ø.564 | 4.2Ø8 | Ø.482 | 8Ø1.491 |
| 1ØØ | 1.Ø16 | 294.716 | 7.282 | 9.Ø89 | Ø.719 | 5.912 | Ø.733 | 1496.855 |
| 125 | 1.248 | 48Ø.923 | 9.654 | 13.69Ø | 1.Ø3Ø | 7.789 | 1.Ø39 | 2454.884 |
| 15Ø | 1.482 | 722.9ØØ | 12.Ø82 | 17.752 | 1.4Ø1 | 9.815 | 1.461 | 37Ø1.577 |
| 175 | 1.714 | 1Ø25.964 | 14.765 | 22.373 | 1.727 | 12.Ø3Ø | 1.922 | 5278.964 |
| 2ØØ | 2.49Ø | 14ØØ.7ØØ | 17.7ØØ | 27.42Ø | 2.Ø19 | 14.396 | 2.481 | 7211.2Ø9 |

of coding, about the effects of small perturbations on the code's efficiency. The questions were submitted to Mr. Dennis Kuba, of CDC in Arden Hills, Minnesota, who is very familiar with STAR FORTRAN. A copy of our letter and his reply is in Appendix C, along with a listing of the routines, their driver, and documentation.

The timings were done using a CPU clock (SECOND) accurate to microseconds. Repeated runs yielded a variation where the last two decimal places ($10^{-6}$ and $10^{-5}$ secs) were random, so the results given in Table 4.11 are averages over all runs and are accurate at least to tenths of milliseconds. The meanings of the various columns are as follows (only CPU times are shown):

1) Number of weights
2) Setup - time to precalculate indices used in implementation
3) Decomposition - time to factor matrix M into form LL*
4) First Back Substitution - time to solve LT = $\overline{S}$
5) Second Back Substitution - time to solve L*W = T
6) Zero Out Sample Covariance Matrix - time to reinitialize the matrix prior to accumulating the covariances
7) Update Sample Covariance Matrix - time to compute and add outer product of one sample vector to matrix
8) Move Sample Covariance Matrix - time to move matrix from one place in core to another (as was done in our implementation, but which is not absolutely necessary; we did it so we could continue to add samples and study the effect of the number of samples on the SNR)
9) TOTAL - time required to completely process 2N samples, where N is the number of weights.

The analysis of Section 3.2 indicates these quantities should be polynomial functions of N, the number of weights. Excluding overhead, we should have

Time for Decomposition + $T_o$ = $(2N^2 - 4N)t_{su}(*)$

$$+(\tfrac{2}{3}N^3 - \tfrac{14}{3}N + 4)t_{op}(*) + 2N\,t_{sop}(*) \qquad 4.13$$

$$+(2N^2 - 6N + 4)t_{su}(\pm) + (\tfrac{2}{3}N^3 - N^2 - \tfrac{11}{3}N + 6)t_{op}(\pm)$$

$$+(2N-2)t_{sop}(\pm) + Nt_{sop}(/) + Nt_{sop}(\sqrt{})$$

Time for First Back Substitution = $T_{FBS}$

$$=(4N-8)t_{su}(*) + (2N^2-2N-4)t_{op}(*) + (2N+4)t_{sop}(*) \qquad 4.14$$

$$+(4N-8)t_{su}(\pm) + (2N^2 - 2N-4)t_{op}(\pm) + 4t_{sop}(\pm)$$

Time for Second Back Substitution = $T_{SBS}$

$$=(4N-8)t_{su}(*) + (2N^2 - 2N-4)t_{op}(*) + (2N+4)t_{sop}(*) \qquad 4.15$$

$$+ (2N-4)t_{su}(SUM) + (2N^2-2N-4)t_{op}(SUM)$$

Time for Updating Matrix = $T_U$

$$=(4N-6)t_{su}(*) + (2N^2-4)t_{op}(\pm) + 4t_{sop}(\pm) \qquad 4.16$$

$$+(4N-6)t_{su}(\pm) + (2N^2-4)t_{op}(\pm) + 4t_{sop}(\pm),$$

where $t_{sop}(\cdot)$ is the time for a scalar operation $(\cdot)$; all times are in milliseconds. Substituting in the above equations the values for $t_{su}(*)$, $t_{op}(*)$, $t_{sop}(*)$, $t_{su}(\pm)$, $t_{op}(\pm)$, and $t_{sop}(\pm)$ from Table 4.10, for the moment approximating SUM values by $\pm$ values, and letting $t_{sop}(/)$ = 1.760E-3 ms and $t_{sop}(\surd)$ = 2.840E-3 ms, we have the following timing predictions in milliseconds:

$$T_D = 10^{-6}(40 \cdot N^3 + 11180 \cdot N^2 - 22380 \cdot N + 11000) \qquad 4.17$$

$$T_{FBS} = 10^{-6}(120 \cdot N^2 + 23080 \cdot N - 42480) \qquad 4.18$$

$$T_{SBS} = 10^{-6}(120 \cdot N^2 + 17480 \cdot N - 32240) \qquad 4.19$$

$$T_U = 10^{-6}(120 \cdot N^2 + 22400 \cdot N - 31280). \qquad 4.20$$

Setup time is a small linear function of N and zero out and move times are small quadratic functions of N. With these insignificant quantities excluded, our theoretical estimate of the total time required to process 2N samples is

$$T_{TOTAL} = 10^{-6}(280 \cdot N^3 + 56220 \cdot N^2 - 44380 \cdot N - 63720). \qquad 4.21$$

Thus although $T_{TOTAL}$ is a cubic function of N, the cubic term does not even equal the quadratic term until N=201, so that the lower-order terms dominate the growth of the function, in particular, the second-from-highest-order terms in all the quantities above.

The form of Equations 4.17 through 4.21 leads one to believe that a log-log plot would extract the dominating term, since

$$y = a\, x^b$$

implies

$$\log y = b \log x + \log a$$

and hence that the graph of log y versus log x should be a straight line whose slope equals the exponent b. If several terms contribute to y, the slope should be close to the exponent in the dominating terms. Figures 4.25 through 4.29 bear out this suspicion. The solid lines indicate the actual times (CPU milliseconds) and the dashed lines the predicted times in Equations 4.17 through 4.21 above as functions of the number of weights, N. As can be seen, both times are almost perfectly straight in all plots, with the predicted line paralleling the actual one. The fact that the two lines are parallel indicates that the analysis accurately predicted the relative magnitudes of the coefficients in the polynomials determining the actual time; but the fact that the predicted time is consistently lower than the actual time indicates that overhead is contributing to the second-highest-order terms, and overhead was not included in the prediction. This overhead results, in part, from the index calculations and from the general overhead involved in using a high-level language like FORTRAN.

That the lines slowly converge as N increases indicates that the floating-point operations in the analysis gradually dominate the overhead terms omitted. Indeed, the actual time lines themselves have a slight downward convexity, indicating an increasing slope and a greater dependence on higher-order terms. This is seen in Table 4.12, which has the logarithmic predictions of the actual time calculated by doing a linear least-squares fit of the data in Figures 4.25 through 4.29. The fit was done to the data as a whole, the data for which $N \leq 30$, and the data for which $N > 30$. The exponents are definitely bigger, and the predicted coefficients are closer to the coefficients in Equations 4.17 through 4.21 when $N > 30$ than when $N \leq 30$.

Table 4.12 Logarithmic Timing Predictions for the CDC STAR-100

| | All Data | | Data for Which N≤30 | | Data for Which N≥30 | |
|---|---|---|---|---|---|---|
| | A | B | A | B | A | B |
| Decomposition | .0169 | 2.13 | .0180 | 2.10 | .0120 | 2.20 |
| First Back Substitution | .0457 | 1.11 | .0531 | 1.05 | .0281 | 1.21 |
| Second Back Substitution | .0388 | 1.21 | .0507 | 1.11 | .0182 | 1.38 |
| Update Sample Covariance Matrix | .0469 | 1.05 | .0673 | .91 | .0222 | 1.22 |
| Total Time to Process 2N Samples | .1368 | 2.02 | .2094 | 1.86 | .0607 | 2.20 |

(Labels "A" and "B" in column labels are constants in the equation $\log_{10}(\text{Time}) = B \log_{10} N + \log_{10} A$ or $\text{Time} = AN^B$.)

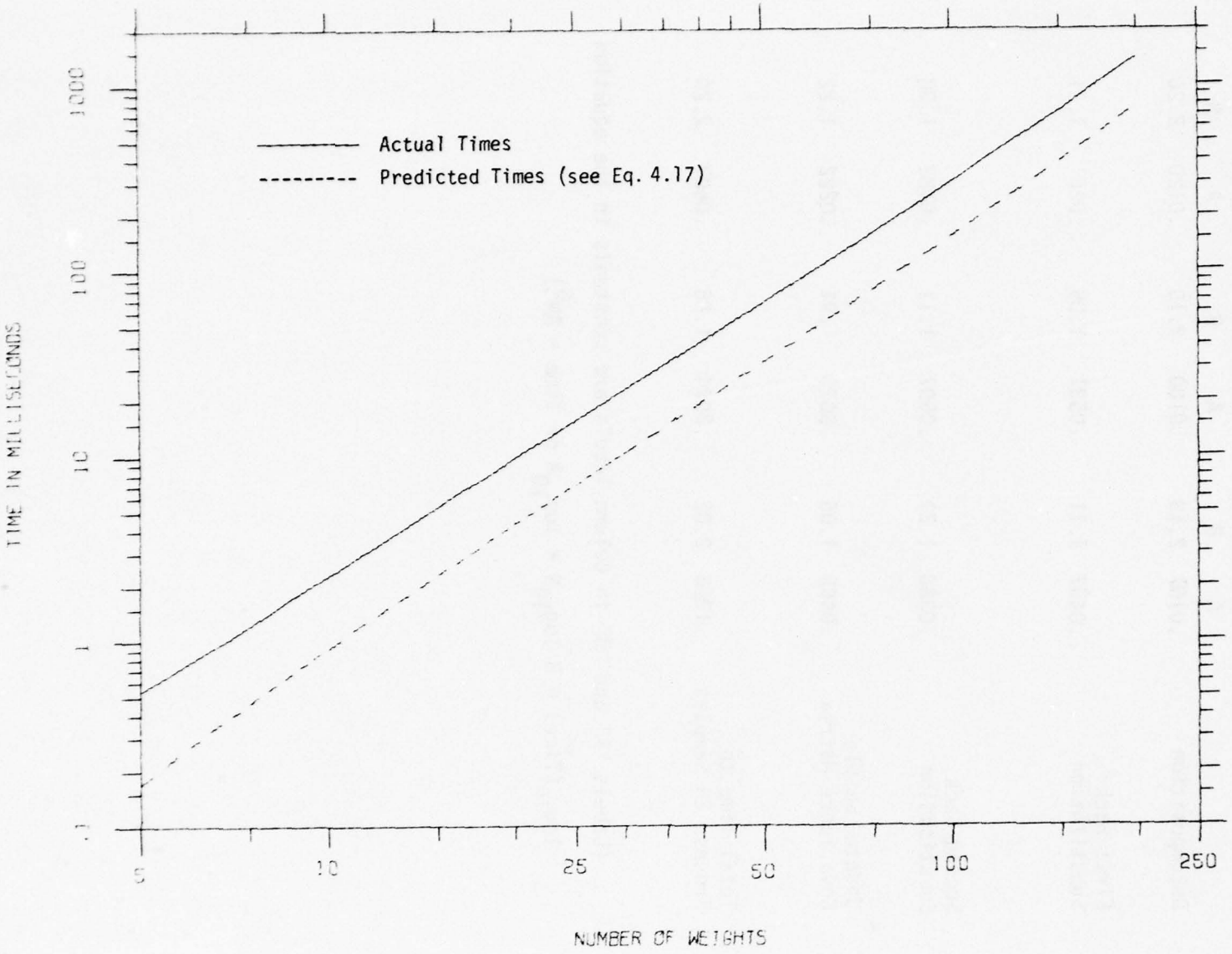



Figure 4.25 CDC STAR-100 Decomposition Times

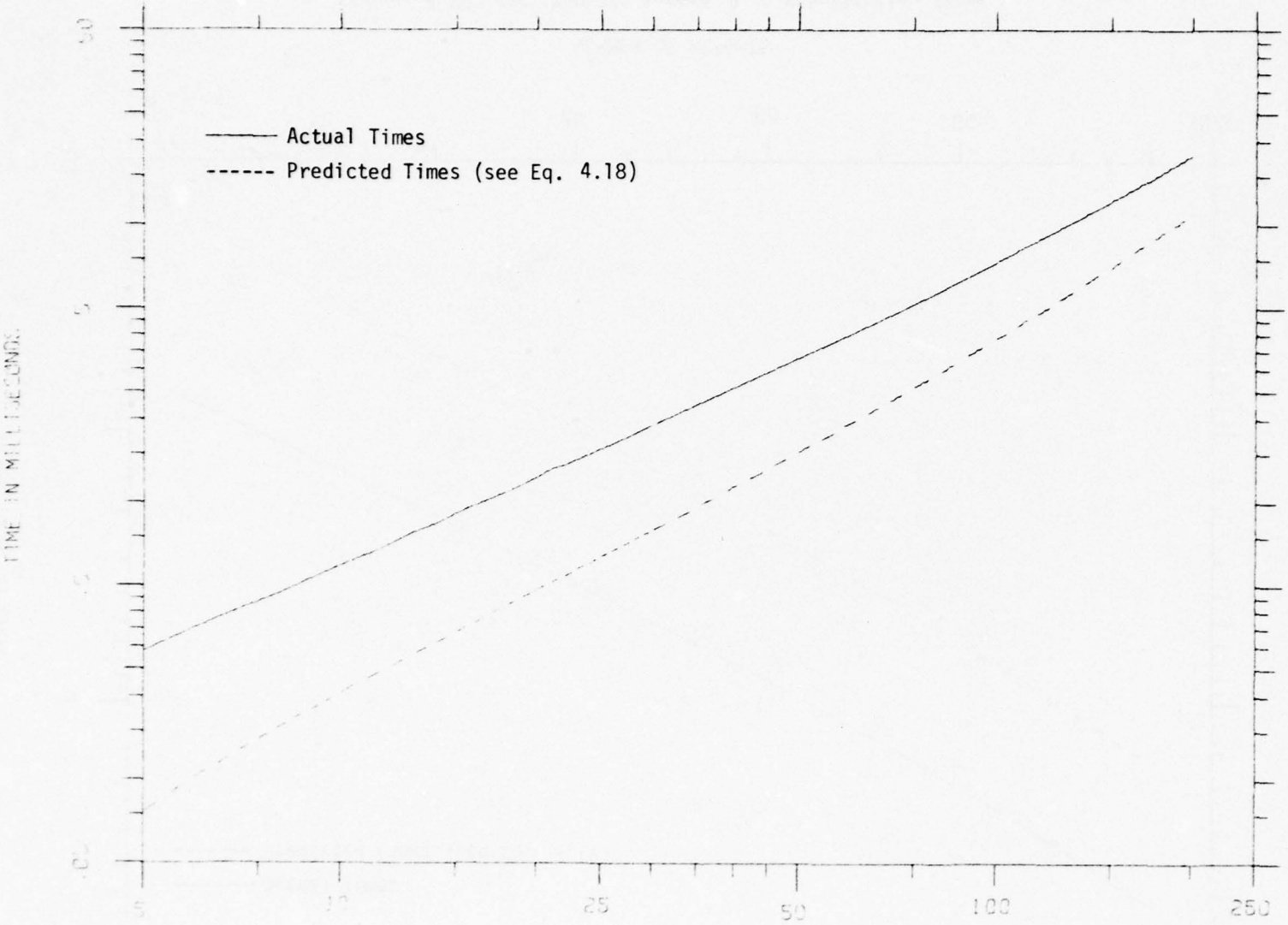



Figure 4.26 CDC STAR-100 First Back Substitution Times

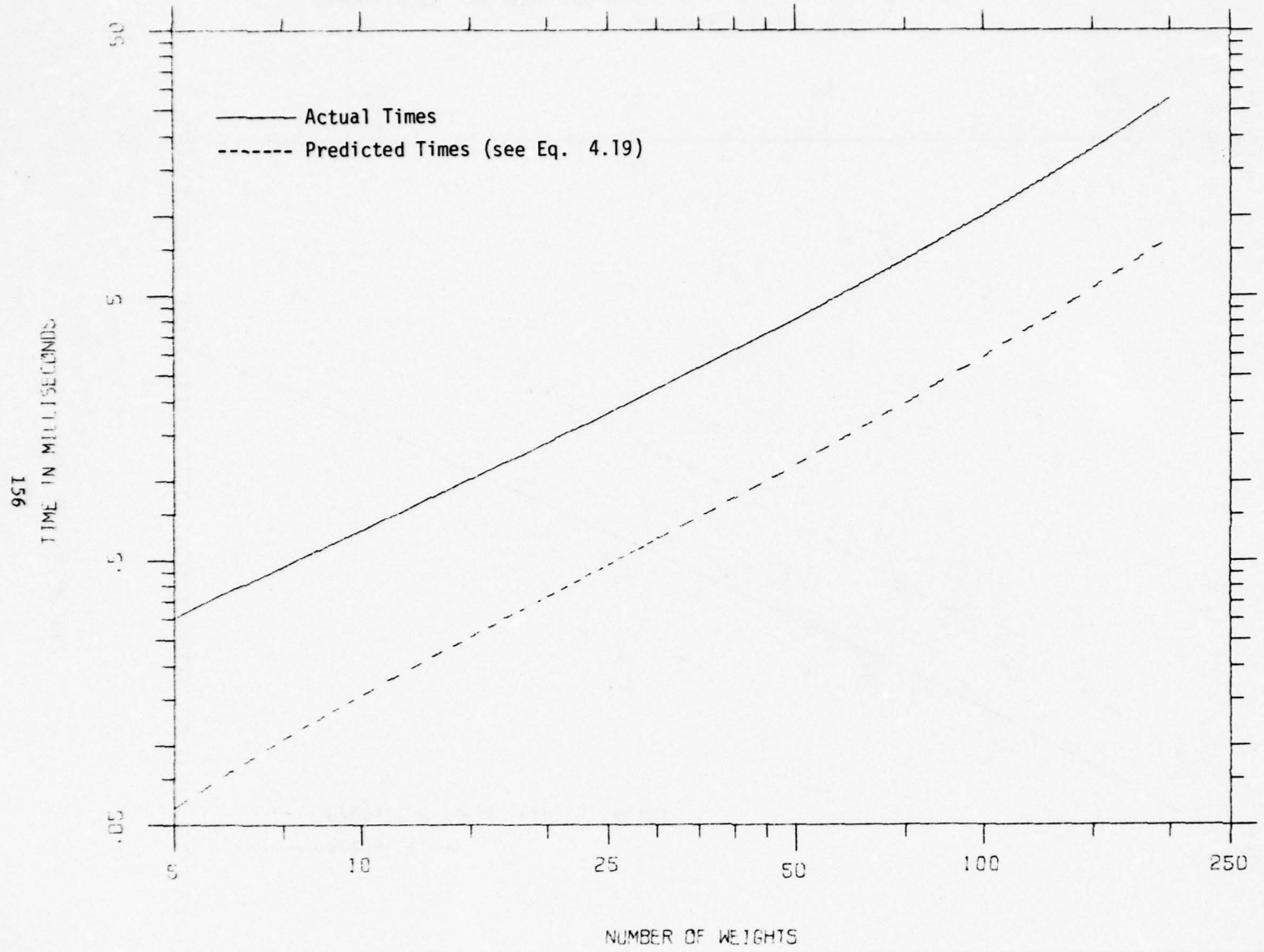



Figure 4.27 CDC STAR-100 Second Back Substitution Times

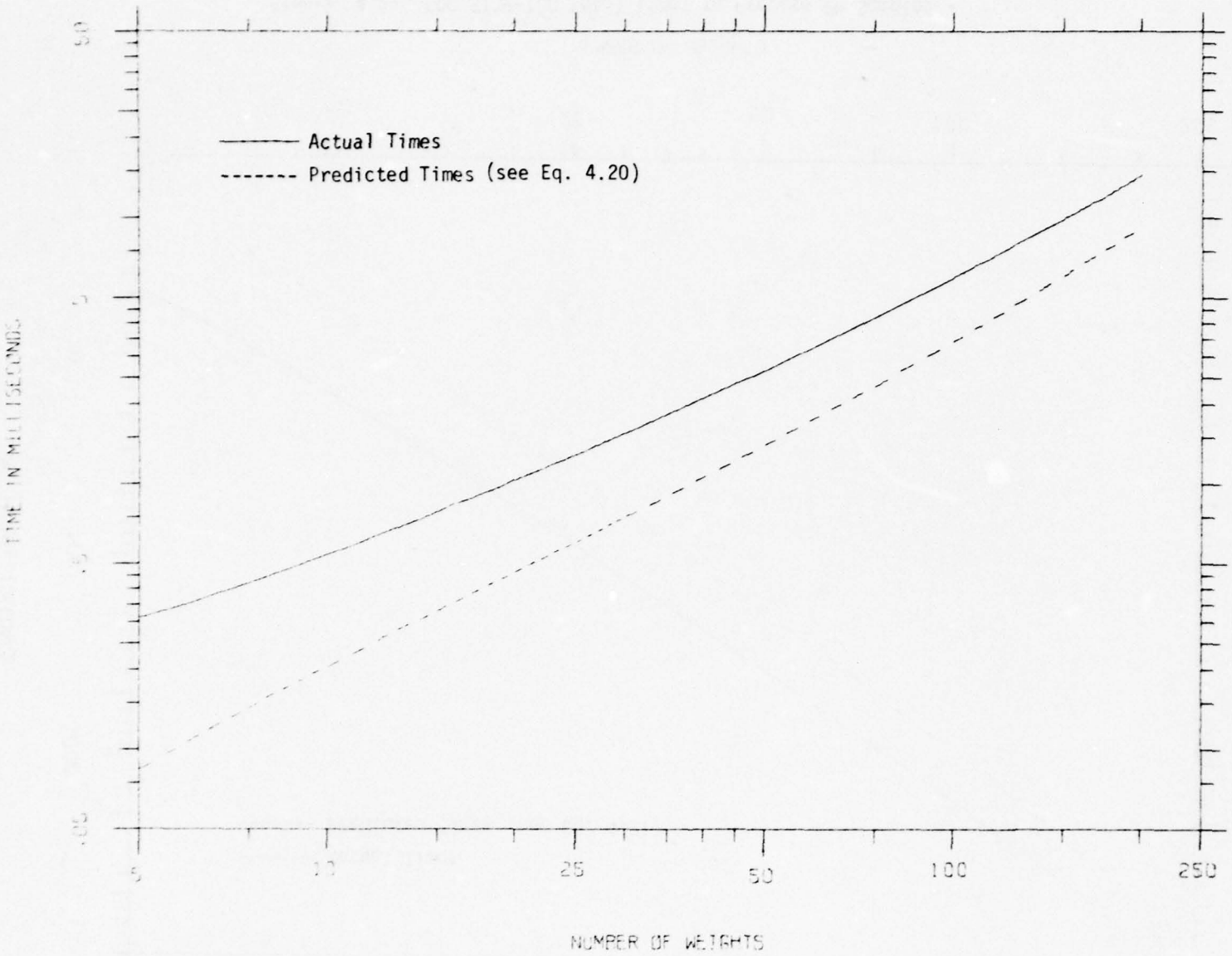



Figure 4.28 CDC STAR-100 Update Sample Covariance Matrix Times

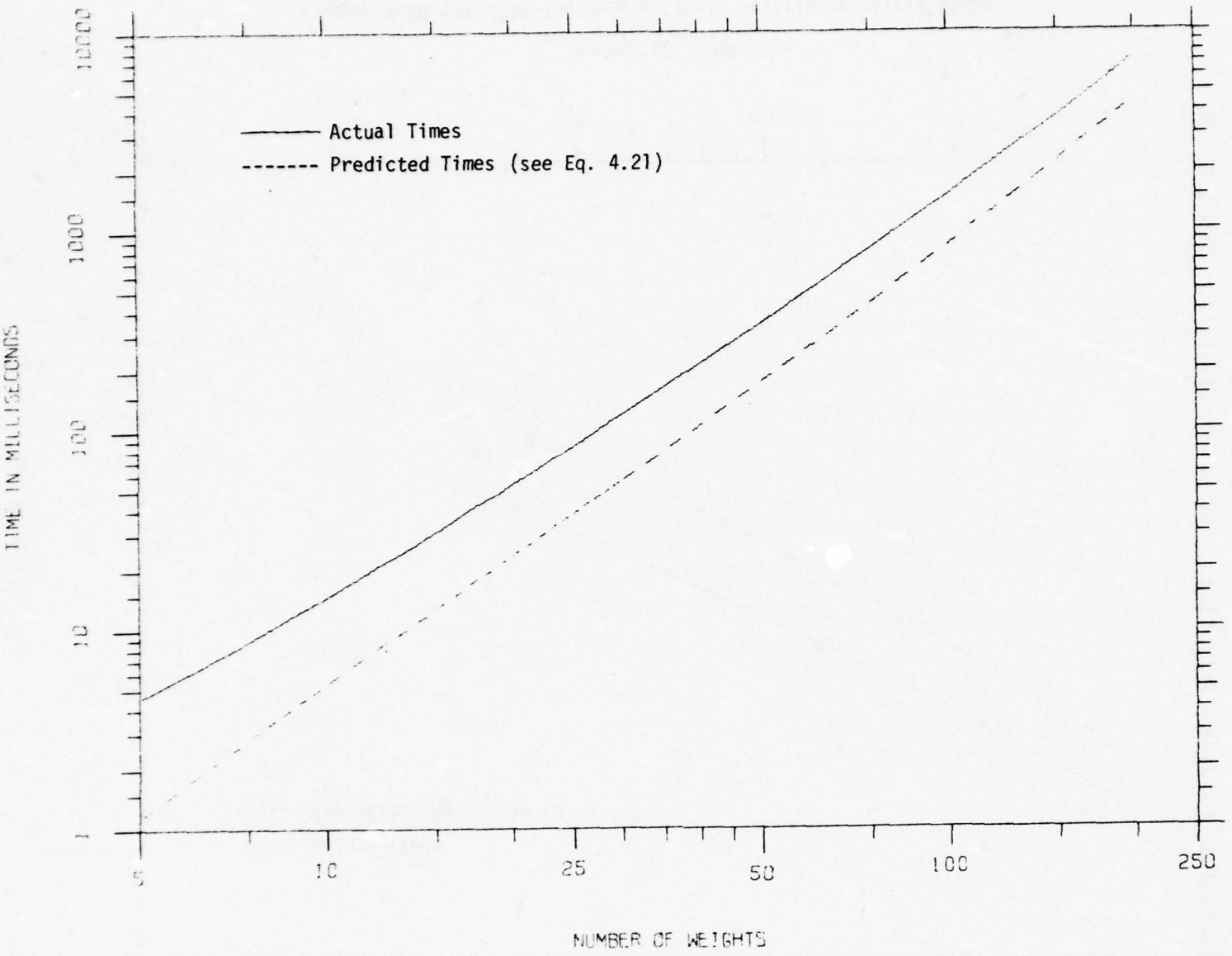



Figure 4.29 CDC STAR-100 Total Times to Process 2N Samples

We also used the model described in Section 2.3 to simulate sample voltage vectors to test the performance of the implementation. We compared the actual SNR achieved to the theoretical prediction of Eq. 2.14 for 200 weights and plotted both values as a function of the number of samples in Figure 4.30. There is obviously a very close agreement between actual and predicted values, which means the sample covariance matrix method is feasible for large numbers of weights.

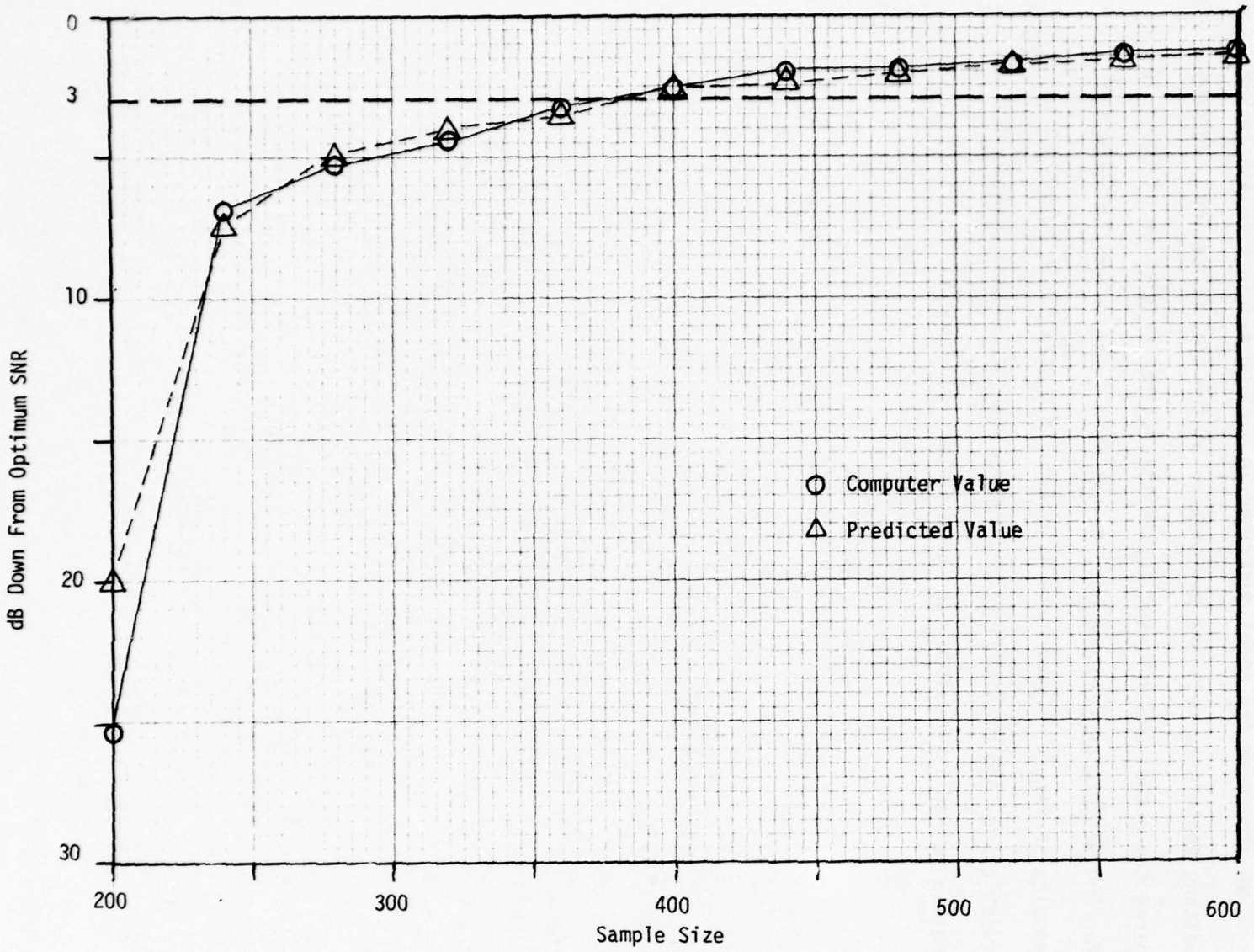



Figure 4.30 Results of Radar Simulation for 200-Weight Adaptive Array Using the Sample Covariance Matrix Inverse Method

4.3.3.2 The CRAY-1

The following introduction to the CRAY-1 was extracted from the CRAY-1 Computer System: Reference Manual [1976] and the CRAY-1 Computer System: An Introduction to Vector Processing [1976].

The CRAY-1 Computer System is a powerful general-purpose computer capable of extremely high processing rates. These rates are achieved by combining scalar and vector capabilities into a single central processor that is joined to a large, fast, bi-polar memory. Vector processing by performing iterative operations on sets of ordered data provides results at rates greatly exceeding result rates of conventional scalar processing. Scalar operations complement the vector capability by providing solutions to problems not readily adapted to vector techniques.

Figure 4.31 represents the basic organization of a CRAY-1 system. The central processor unit (CPU) is a single integrated processing unit consisting of a computation section, a memory section, and an input/output section. The memory is expandable from 0.25 million 64-bit words to a maximum of 1.0 million words. The 12 full duplex I/O channels in the input/output section connect to a maintenance control unit (MCU), a mass storage subsystem, and a variety of I/O stations or peripheral equipment. The MCU provides for system initialization and for monitoring system performance. The mass storage subsystem provides secondary storage and consists of one to eight Cray Research Inc. DCU-2 Disk Controllers, each with one to four DD-19 Disk Storage Units. Each DD-19 has a capacity of $2.424 \times 10^9$ bits so that a maximum mass storage configuration could hold $9.7 \times 10^9$ 8-bit characters.

I/O channels can be connected to independent processors referred to as I/O stations or to peripheral equipment according to the requirements of the

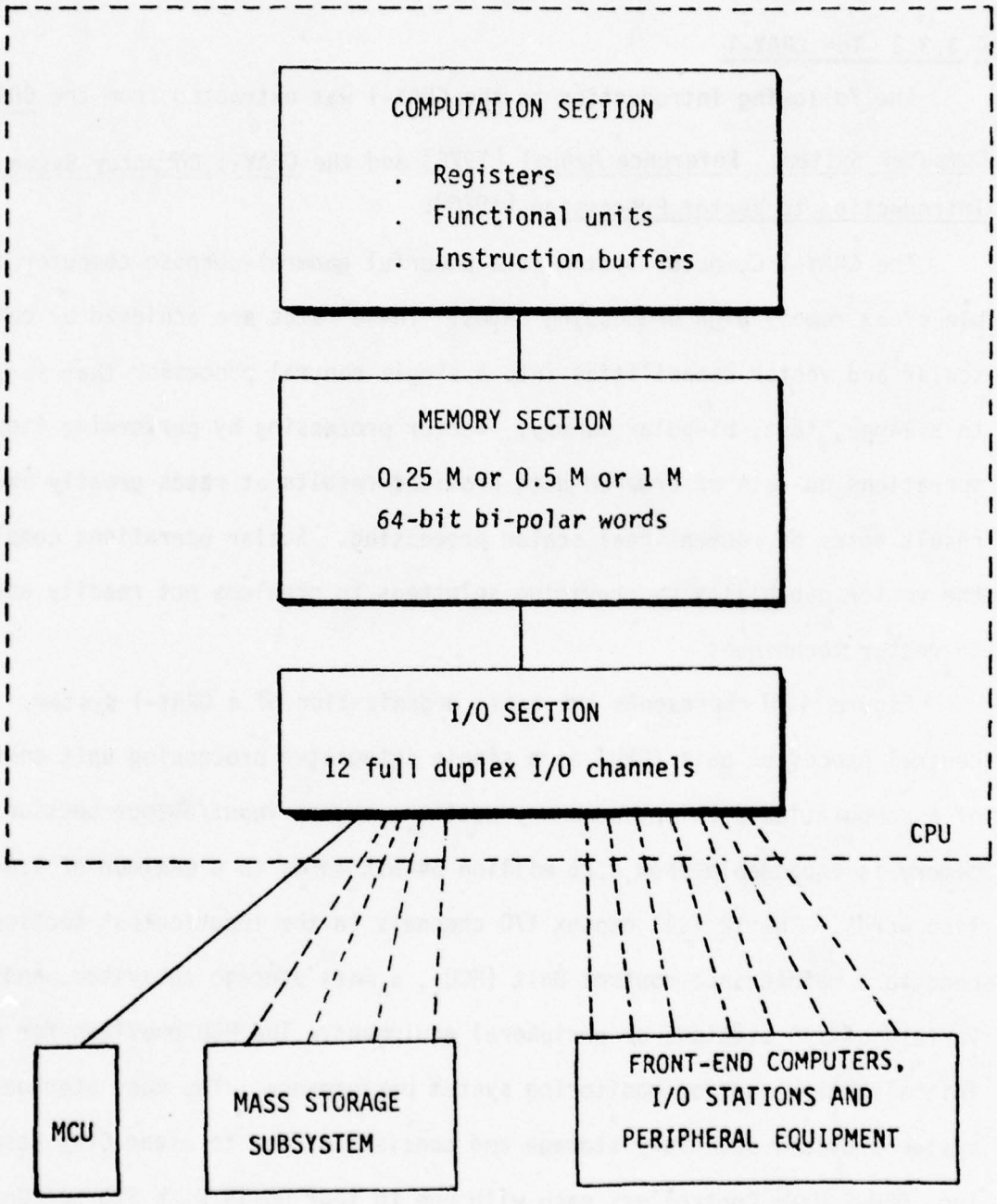



Figure 4.31 Basic CRAY-1 Organization

individual installation. At least one I/O station is considered standard to collect data and present it to the CRAY-1 for processing and to receive output from the CRAY-1 for distribution to slower devices.

Table 4.13 summarizes the characteristics of the system. The following paragraphs provide an additional introduction to the three sections of the CPU.

The computation section contains instruction buffers, registers and functional units which operate together to execute a program of instructions stored in memory.

Arithmetic operations are either integer or floating-point. Integer arithmetic is performed in two's complement mode. Floating-point quantities have signed-magnitude representation.

The CRAY-1 executes 128 operation codes as either 16-bit (one-parcel) or 32-bit (two-parcel) instructions. Operation codes provide for both scalar and vector processing.

Floating-point instructions provide for addition, subtraction, multiplication, and reciprocal approximation. The reciprocal approximation instruction allows for the computation of a floating divide operation using a multiple instruction sequence.

Integer or fixed-point operations are provided as follows: integer addition, integer subtraction, and integer multiplication. An integer multiply operation produces a 24-bit result; additions and subtractions produce either 24-bit or 64-bit results. No integer divide instruction is provided, and the operation is accomplished through a software algorithm, using floating-point hardware.

Table 4.13 Characteristics of the CRAY-1 Computer System

COMPUTATION SECTION

- 64-bit word
- 12.5 nanosecond clock period
- 2's complement arithmetic
- Scalar and vector processing modes
- Twelve fully segmented functional units
- Eight 24-bit address (A) registers
- Sixty-four 24-bit intermediate address (B) registers
- Eight 64-bit scalar (S) registers
- Sixty-four 64-bit intermediate scalar (T) registers
- Eight 64-element vector (V) registers, 64-bits per element
- Four instruction buffers of 64 16-bit parcels each

MEMORY SECTION

- Up to 1,048,576 words of bi-polar memory
  (64 data bits and one parity bit per word)
- Sixteen banks of up to 65,536 words each
- Four-clock-period bank cycle time
- One-word-per-clock-period transfer rate to B, T, and V registers
- One-word-per-two-clock-periods transfer rate to A and S registers
- Four-words-per-clock-period transfer rate to instruction buffers

INPUT/OUTPUT SECTION

- Twelve full duplex I/O channels
- Channel groups contain either six input or six output channels
- Channel groups served equally by memory (scanned every four clock periods)
- Channel priority resolved within channel groups
- Sixteen data bits and three control bits per channel
- Lost data detection

The instruction set includes Boolean operations for OR, AND, and exclusive OR and for a mask-controlled merge operation. Shift operations allow the manipulation of either 64-bit or 128-bit operands to produce 64-bit results. With the exception of 24-bit integer arithmetic, all operations are implemented in vector as well as scalar instructions. The integer product is a scalar instruction designed for index calculation. Full indexing capability allows the programmer to index throughout memory in either scalar or vector modes. The index may be positive or negative in either mode, thereby allowing matrix operations in vector mode to be performed on rows or the diagonal as well as conventional column-oriented operations.

Each functional unit implements an algorithm or a portion of the instruction set. Units are independent and are fully segmented. This means that a new set of operands for unrelated computation may enter a functional unit each clock period.

All operands processed by the CRAY-1 are held in registers prior to their being processed by the functional units and are received by registers after processing. In general, the sequence of operations is to load one or more vector registers from memory and pass them to functional units. Results from this operation are received by another vector register and may be processed additionally in another operation or returned to memory if the results are to be retained.

The contents of a V register are transferred to or from memory by specifying a first-word address in memory, an increment for the memory address, and a length. The transfer proceeds beginning with the first element of the V register and incrementing by one in the V register at a rate of up to one word per clock period, depending on memory conflicts.

A result may be received by a V register and re-entered as an operand to another vector computation in the same clock period. This mechanism allows for "chaining" two or more vector operations together. Chain operation allows the CRAY-1 to produce more than one result per clock period. Chain operation is detected automatically by the CRAY-1 and is not explicitly specified by the programmer, although the programmer may reorder certain code segments in order to enable chain operation.

There may be a conflict between scalar and vector operations only for the floating-point operations and storage access. With the exception of these operations, the functional units are always available for scalar operations. A vector operation will occupy the selected functional unit until the vector has been processed.

Parallel vector operations may be processed in two ways:

1) Using different functional units and all different V registers.
2) Chain mode, using the result stream from one vector register simultaneously as the operand to another operation using a different functional unit.

Parallel operations on vectors allow the generation of two or more results per clock period. Most vector operations use two vector registers as operands or one scalar and one vector register as operands. Exceptions are

segments; however, a programmer may choose to construct the vector loop code in any of a number of ways. The processing of long vectors in FORTRAN is handled by the compiler and is transparent to the programmer.

There are twelve functional units in the CPU. Each is a specialized unit implementing algorithms for a portion of the instructions. Each unit is independent of the other units and a number of functional units may be in operation at the same time. A functional unit receives operands from registers and delivers the result to a register when the function has been performed. There is no information retained in a functional unit for reference in subsequent instructions. These units operate essentially in three-address mode with limited source and destination addressing.

Three functional units provide 24-bit results to the A registers only:

- Integer add
- Integer multiply
- Population count.

Three functional units provide 64-bit results to the S registers only:

- Integer add
- Shift
- Logical.

Three functional units provide 64-bit results to the V registers only:

- Integer add
- Shift
- Logical.

Three functional units provide 64-bit results to either the S or V registers:

- Floating add
- Floating multiply
- Reciprocal approximation

Integer arithmetic is performed in 2's complement mode. Floating-point quantities have signed-magnitude representation.

All functional units are fully segmented. This means that the information arriving at the unit, or moving within the unit, is captured and held in a new set of registers at the end of every clock period. It is therefore possible to start a new set of operands for unrelated computation into a functional unit each clock period even though the unit may require more than one clock period to complete the calculation. All functional units perform their algorithms in a fixed amount of time. No delays are possible once the operands have been delivered to the unit. Functional units servicing the vector instructions produce one result per clock period.

Floating-point numbers are represented in a standard format throughout the CPU. This format is a packed representation of a binary coefficient and an exponent or power of two. The coefficient is a 48-bit signed fraction. The sign of the coefficient is separated from the rest of the coefficient as shown in Figure 4.32. Since the coefficient is signed magnitude, it is not complemented for negative values.

The exponent portion of the floating-point format is represented as a biased integer in bits 1 through 15. The bias that is added to the exponents is $40000_8$. The positive range of exponents is $40000_8$ through $57777_8$. The negative range of exponents is $37777_8$ through $20000_8$. Thus, the unbiased range of exponents is the following:

$$2^{-17777_8} \text{ to } 2^{+17777_8} .$$

In terms of decimal values, the floating-point format of the CRAY-1 allows the expression of numbers accurate to about 15 decimal digits in the approximate decimal range of $10^{-2500}$ through $10^{+2500}$.

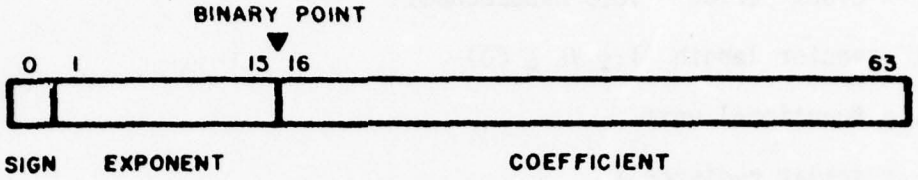



Figure 4.32 CRAY-1 Floating-Point Data Format

Table 4.14 contains the timings for various scalar and vector operations on the CRAY-1. Note that instructions 176 and 177 permit non-unity address increments, allowing rowwise and columnwise access of matrices.

The memory for the CRAY-1 consists of 16 banks of bi-polar 1024-bit LSI memory. Three memory size options are available: 262,144 words, 524,288 words, or 1,048,576 words. Each word is 65 bits including one parity bit. The banks are independent of each other.

Sequentially addressed words reside in sequential banks. The memory cycle time is four clock periods (50 nsec). The access time, that is, the time required to fetch an operand from memory to a scalar register, is 10 clock periods (125 nsec). There is no inherent memory degradation for machines with less than one million words of memory.

The maximum transfer rate for B, T, and V registers is one word per clock period. For A and S registers, it is one word per two clock periods. Transfers of instructions to the instruction buffers occur at a rate of 16 parcels (four words) per clock period.

Table 4.14 CRAY-1 Timings for Various Scalar and Vector Operations

Abbreviations

CP = clock period = 12.5 nanoseconds

VL = vector length ($1 \leq VL \leq 64$)

f.u. = functional unit

$S_j$ = scalar register j

$V_j$ = vector register j

FP = floating point

Instructions

062 - scalar FP sum of $S_i$ and $S_j$ to $S_k$

063 - scalar FP difference of $S_i$ and $S_j$ to $S_k$

064 - scalar FP products of $S_i$ and $S_j$ to $S_k$

065 - scalar half-precision rounded FP product of $S_i$ and $S_j$ to $S_k$

066 - scalar FP rounded FP products of $S_i$ and $S_j$ to $S_k$

067 - scalar FP reciprocal iteration; $2-S_i*S_j$ to $S_k$

070 - scalar FP reciprocal of $S_i$ to $S_j$

160 - vector FP products of $S_i$ and $V_j$ to $V_k$

161 - vector FP products of $V_i$ and $V_j$ to $V_k$

162 - vector half-precision rounded FP products of $S_i$ and $V_j$ to $V_k$

163 - vector half-precision rounded FP products of $V_i$ and $V_j$ to $V_k$

164 - vector rounded FP products of $S_i$ and $V_j$ to $V_k$

165 - vector rounded FP products of $V_i$ and $V_j$ to $V_k$

166 - vector FP reciprocal iterations; $2-S_i*V_j$ to $V_k$

167 - vector FP reciprocal iteration; $2-V_i*V_j$ to $V_k$

170 - vector FP sums of $S_i$ and $V_j$ to $V_k$

Table 4.14 CRAY-1 Timings for Various Scalar and Vector Operations (Cont'd)

171 - vector FP sums of $V_i$ and $V_j$ to $V_k$

172 - vector FP differences of $S_i$ and $V_j$ to $V_k$

173 - vector FP differences of $V_i$ and $V_j$ to $V_k$

174 - vector FP reciprocal approximation of $V_i$ to $V_j$

176 - transmit VL words from memory to $V_i$

177 - transmit VL words from $V_i$ to memory

FUNCTIONAL UNIT TIMING

| | |
|---|---|
| Address integer add | 2 CP |
| Address integer multiply | 6 CP |
| Scalar integer add | 3 CP |
| Scalar logical | 1 CP |
| Scalar shift | |
| single shift | 2 CP |
| double shift | 3 CP |
| Population/leading zero count | |
| leading zero count | 3 CP |
| population | 4 CP |
| Vector integer add | 3 CP |
| Vector logical | 2 CP |
| Vector shift | 4 CP |
| Floating-point add | 6 CP |
| Floating-point multiply | 7 CP |
| Floating-point reciprocal | 14 CP |

Table 4.14 CRAY-1 Timings for Various Scalar and Vector Operations (Cont'd)

**VECTOR TIMING**

**Issue on chain slot:**

Chain slot issue = vector f.u. time + 2 CP

The chain slot issue can occur only during the above CP.
Both operands must be available during this CP.

**Issue on register i free:**

Issue if (VL) > 5 = vector f.u. time + (VL) + 2 CP
Issue if (VL) $\leq$ 5 = vector f.u. time + 7 CP

**Issue on unit free:**

Memory to or from vector register: 176 = (VL) + 4 CP; 177 = (VL) + 5 CP
Vector register to vector register = (VL) = 4 CP

Issue on operand register free:

Issue if (VL) > 5 = (VL)
Issue if (VL) $\leq$ 5 = 5 CP

SCALAR TIMING

Issue on register i free:

Issue = scalar f.u. time

Issue of scalar instructions using floating-point functional units:

Issue 170, 172, 062 - 063 preceded by 170 - 173 = (VL) + 4 CP
Issue 160, 162, 164, 166, 064 - 067 preceded by 160 - 167 = (VL) + 4 CP
Issue 070 preceded by 174 = (VL) + 4 CP

Issue on scalar register conflict:

Issue is delayed 1 CP if a result register conflict would exist with a previously issued instruction.

Thus, the high speed of memory supports the requirements of scientific applications while its low cycle time is well-suited to random access applications. The phased memory banks allow high communication rates through the I/O section and provide low read/store times for vector registers.

The CRAY-1 is currently being updated and an improved version, the CRAY-2, will become available in the next few years.

We chose the same algorithms for updating the sample covariance matrix, decomposition, and back substitution as on the STAR and for the same reasons as on the STAR, with one exception. The CRAY-1 does not have a hardware square root, and so $t_{sop}(\sqrt{\ }) = 1762.5$ ns, which is large, but LL* still becomes superior to LDL* by the time N exceeds 140, so LL* was retained. Hence Equations 4.13 through 4.16 are still valid, but only as long as $N \leq 64$, because the maximum vector length on the CRAY-1 is 64, and longer vectors must be broken up, resulting in longer startup times. So, substituting values from Table 4.14 in Eqs. 4.13 through 4.16 and approximating SUM times by $\pm$ times (as on the STAR) we obtain the following timing predictions in milliseconds:

$$T_D = 10^{-6}(16.67*N^3 + 412.5*N^2 + 1108.33*N + 375) + E_D(N) \qquad 4.22$$

$$T_{FBS} = 10^{-6}(50*N^2 + 975*N - 1150) + E_{FBS}(N) \qquad 4.23$$

$$T_{SBS} = 10^{-6}(50*N^2 + 775*N - 1050) + E_{SBS}(N) \qquad 4.24$$

$$T_U = 10^{-6}(50*N^2 + 850*N - 725) + E_u(N) \qquad 4.25$$

$$T_{TOTAL} = 10^{-6}(116.67*N^3 + 2212.5*N^2 + 1408.33*N \qquad 4.26$$

$$- 1825) + E_{TOTAL}(N) \ ,$$

where $E_D(N)$, $E_{FBS}(N)$, $E_{SBS}(N)$, $E_U(N)$, and $E_{TOTAL}(N)$ are the errors in startup time made by ignoring the extra startups when N > 64 (they are all zero when N = 64).

The actual timings of the program were done using the system routine SECOND: CALL SECOND(T) returns the number of CPU seconds used by the program in the floating-point variable T. Table 4.15 contains the CPU times in milliseconds. The meanings of the various columns are as follows:

1) Number of weights
2) Decomposition - time to factor matrix M into form LL*
3) First Back Substitution - time to solve LT = $\bar{S}$ for T
4) Second Back Substitution - time to solve L*W = T for W
5) Update Sample Covariance Matrix - time to compute and add the outer product of one sample vector to the sample covariance matrix
6) Total Time - time to form the sample covariance matrix from 2N samples and solve for W with decomposition and two back substitutions.

To compare actual and predicted times, as with the STAR and CDC 7600, we plot actual and predicted times on a log-log plot, using a solid line for the actual times from Table 4.15 and a dashed line for the predicted times from Eqs. 4.22 through 4.26. These plots are found in Figs. 4.33 through 4.37. As with the CDC STAR-100, the lines on the plots are essentially parallel, indicating that the prediction equations provide the correct rates of growth of the times as functions of N, the number of

## Table 4.15 CRAY-1 CPU Times in Milliseconds

| N = # Weights | Decomposition | First Back Substitution | Second Back Substitution | Update Sample Covariance Matrix | Total Time |
|---|---|---|---|---|---|
| 20 | 1.922 | .125 | .258 | .101 | 6.419 |
| 25 | 3.537 | .162 | .384 | .134 | 10.859 |
| 30 | 5.877 | .202 | .535 | .170 | 16.971 |
| 35 | 9.083 | .245 | .712 | .209 | 24.776 |
| 40 | 13.290 | .291 | .935 | .251 | 34.714 |
| 45 | 18.631 | .341 | 1.145 | .296 | 46.948 |
| 50 | 25.245 | .393 | 1.397 | .353 | 62.545 |
| 60 | 42.836 | .508 | 1.983 | .451 | 93.771 |
| 70 | 67.153 | .639 | 2.672 | .577 | 151.564 |
| 80 | 99.280 | .794 | 3.462 | .720 | 230.326 |
| 90 | 141.315 | .954 | 4.359 | .876 | 328.772 |
| 100 | 194.333 | 1.134 | 5.354 | 1.044 | 457.319 |
| 110 | 258.427 | 1.319 | 6.455 | 1.224 | 621.223 |
| 120 | 327.713 | 1.520 | 7.659 | 1.417 | 877.048 |
| 130 | 415.283 | 1.734 | 8.966 | 1.625 | 969.524 |
| 140 | 517.128 | 1.972 | 10.375 | 1.856 | 1,149.269 |
| 150 | 634.474 | 2.223 | 11.888 | 2.097 | 1379.695 |
| 160 | 768.249 | 2.486 | 13.500 | 2.352 | 1633.224 |
| 170 | 919.754 | 2.762 | 15.223 | 2.619 | 1920.375 |
| 180 | 1089.918 | 3.050 | 17.043 | 2.899 | 2235.424 |
| 190 | 1279.871 | 3.350 | 18.966 | 3.191 | 2576.952 |
| 200 | 1490.769 | 3.671 | 20.993 | 3.505 | 2950.241 |

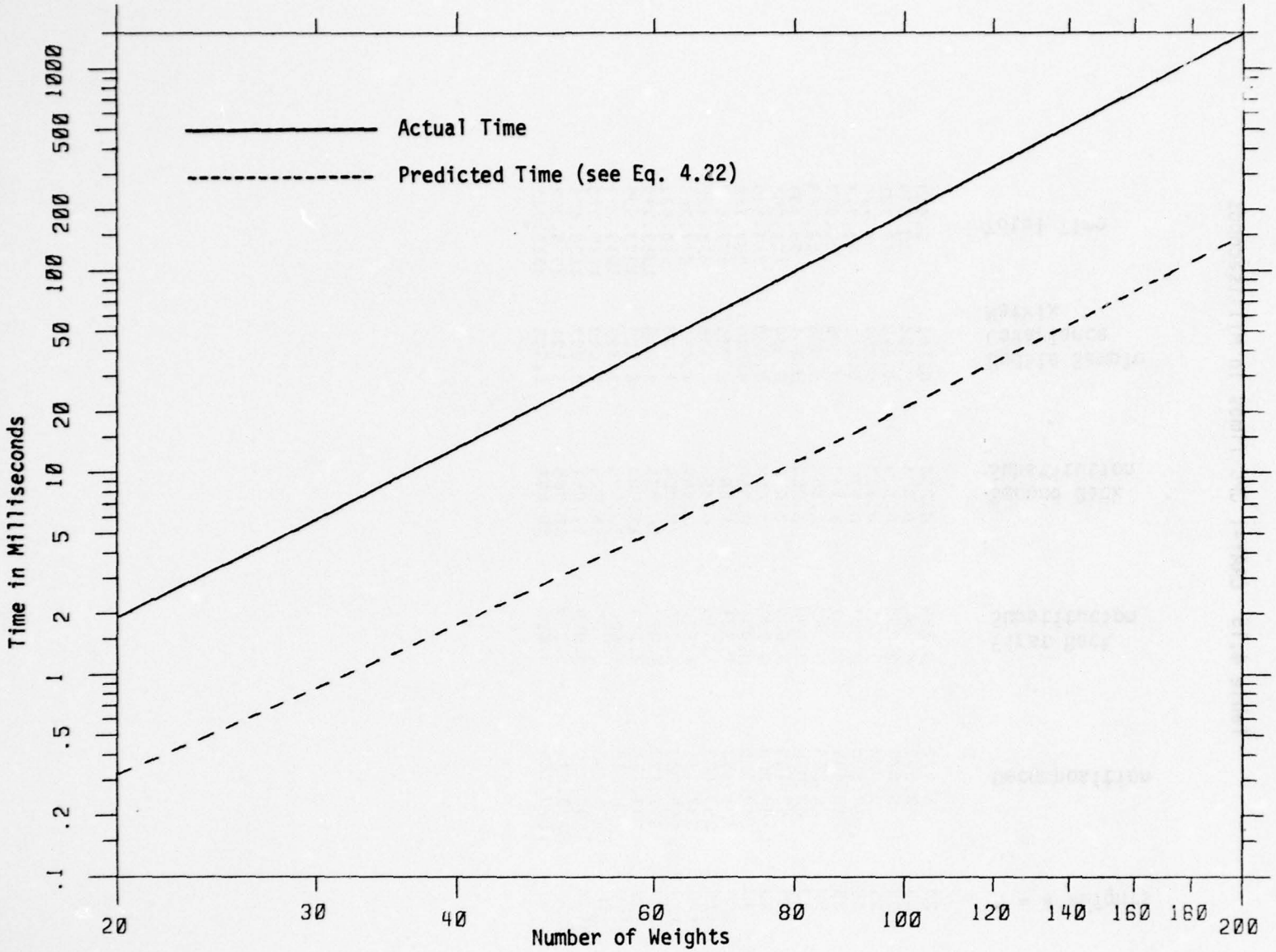



Figure 4.33 CRAY-1 Decomposition Times

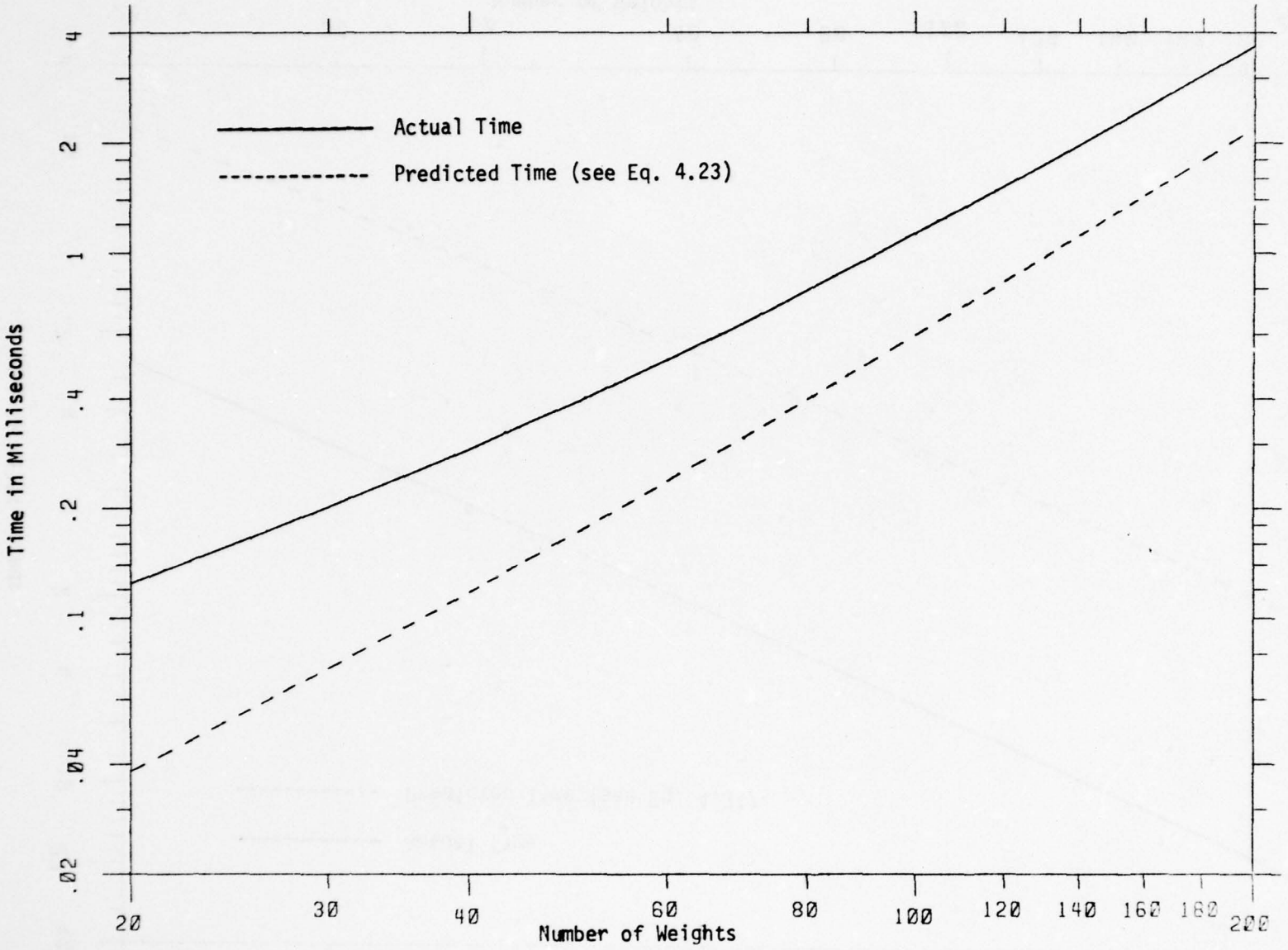



Figure 4.34 CRAY-1 First Back Substitution Times

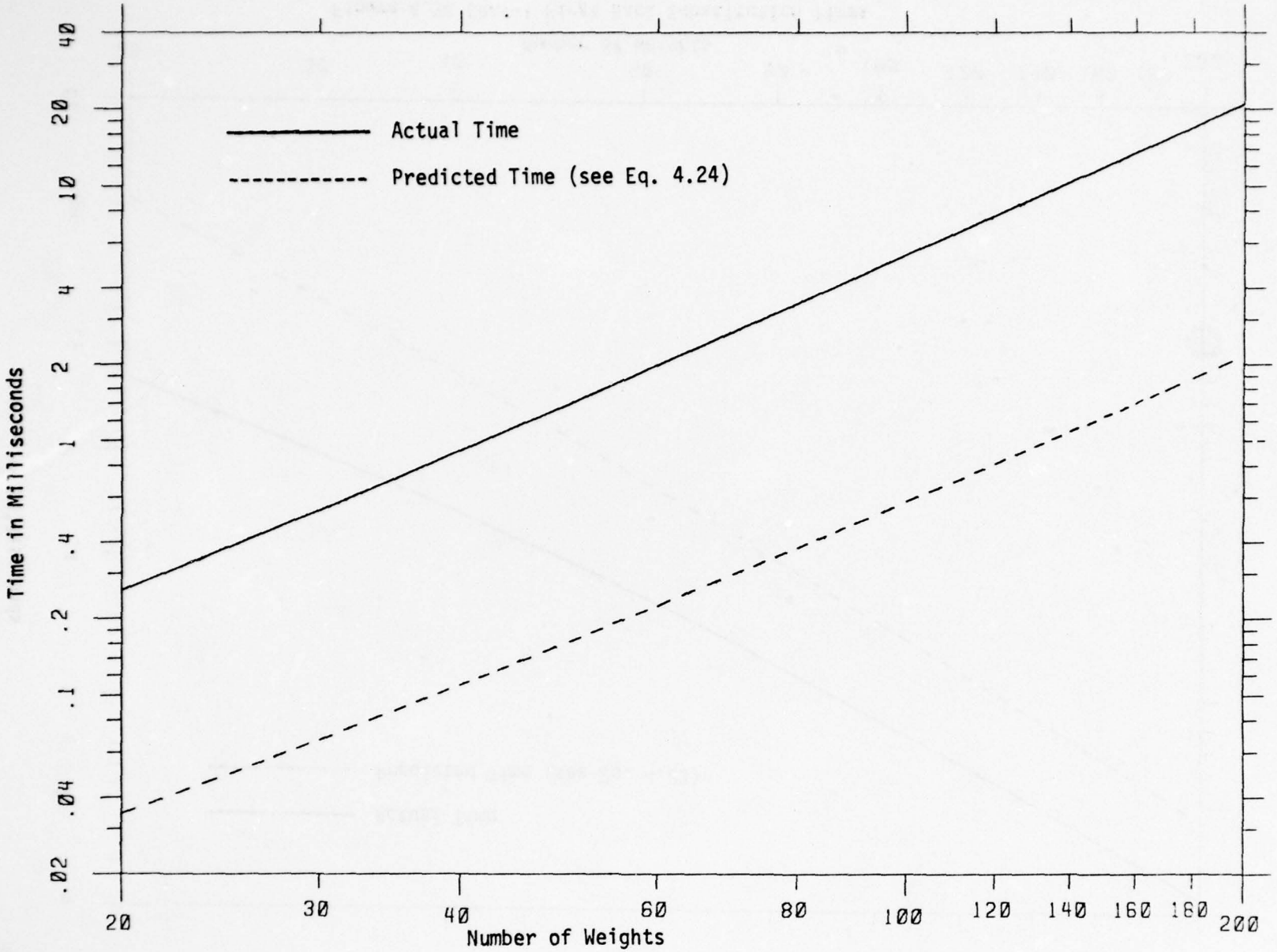



*Figure 4.35* CRAY-1 Second Back Substitution Times

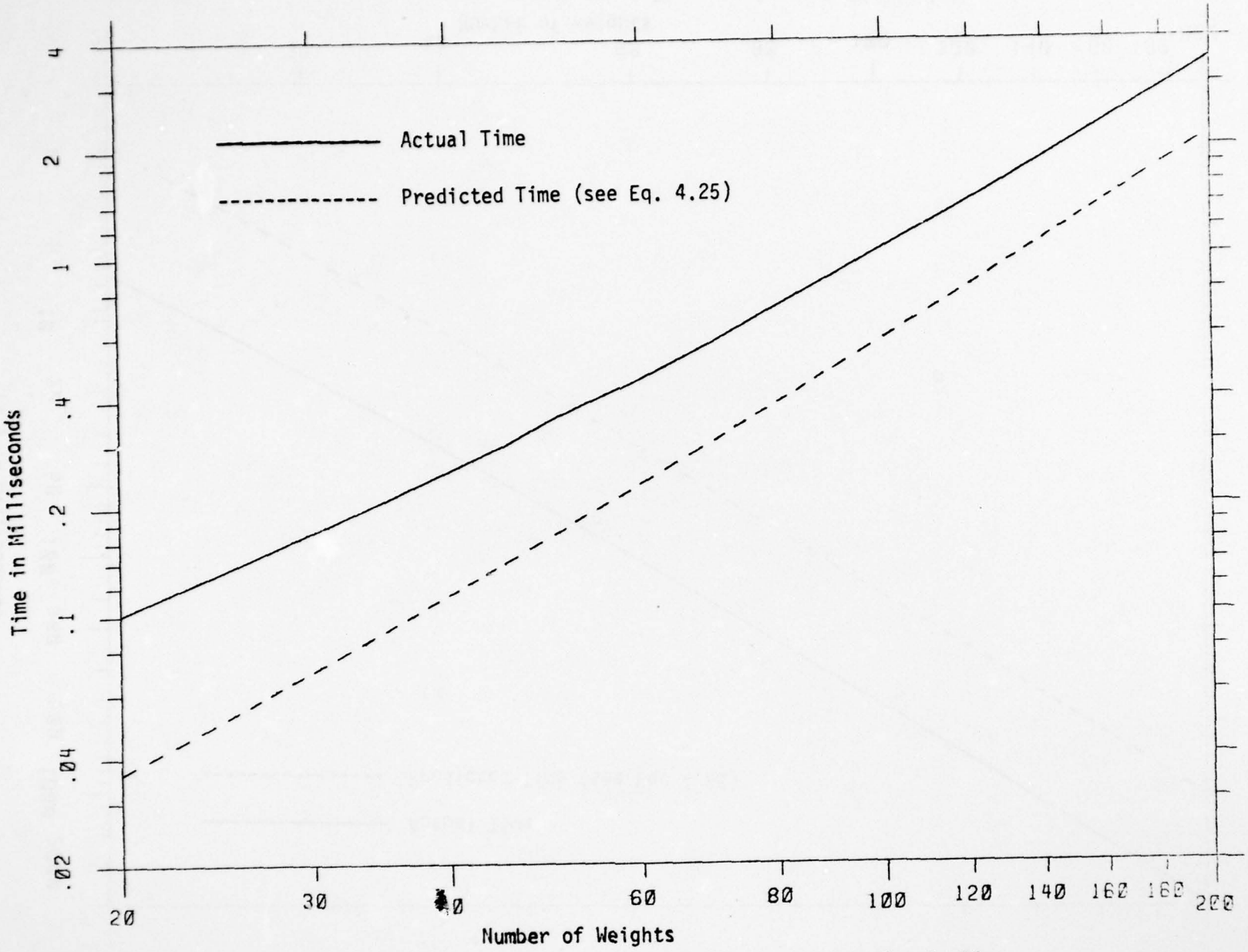
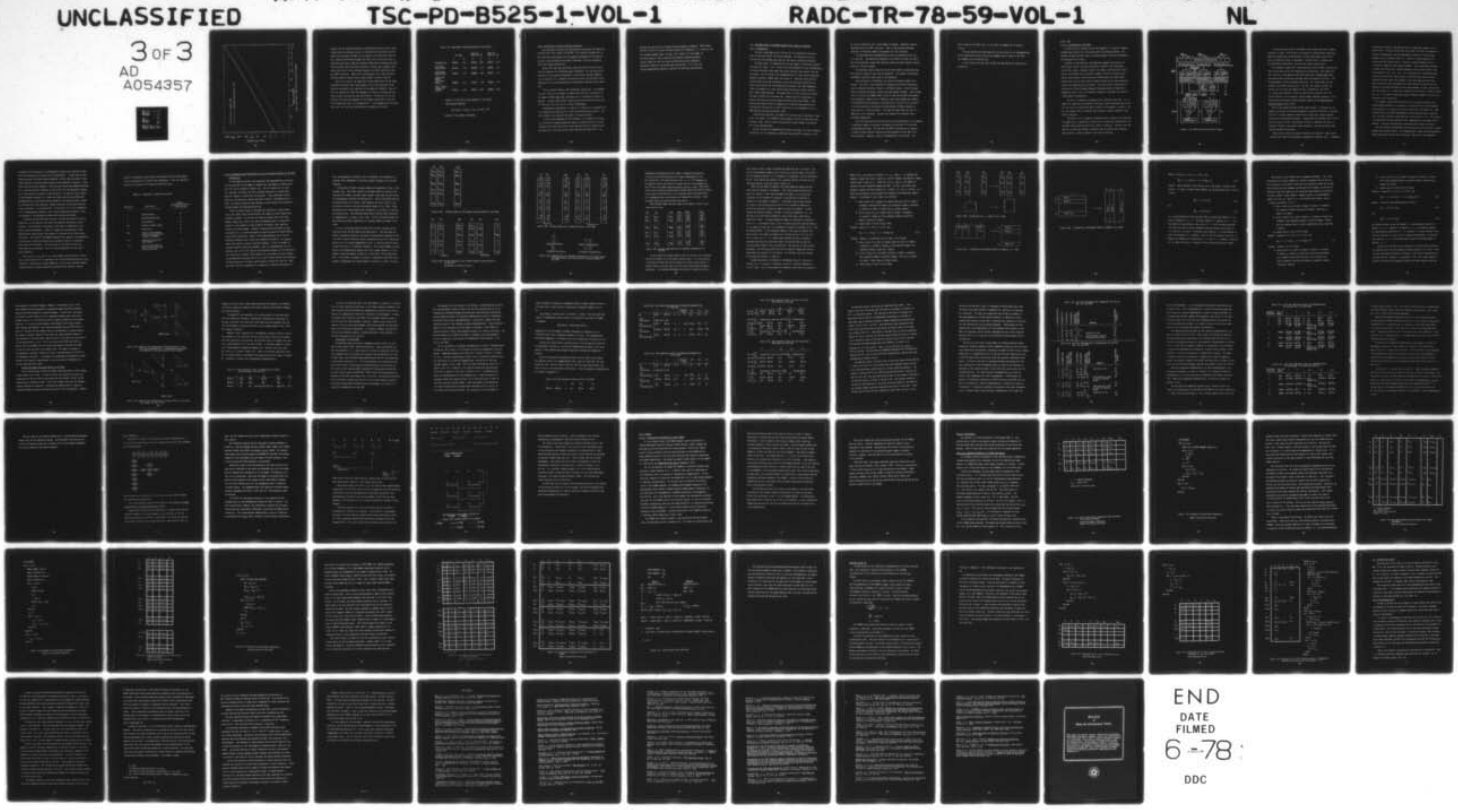



Figure 4.36 CRAY-1 Update Sample Covariance Matrix Time

AD-A054 357 TECHNOLOGY SERVICE CORP SANTA MONICA CALIF F/G 17/9
MULTIDOMAIN ALGORITHM EVALUATION. VOLUME I.(U)
APR 78 W C LILES, J C DEMMEL, I S REED F30602-76-C-0319
UNCLASSIFIED TSC-PD-B525-1-VOL-1 RADC-TR-78-59-VOL-1 NL

3 OF 3
AD
A054357



END
DATE
FILMED
6-78
DDC

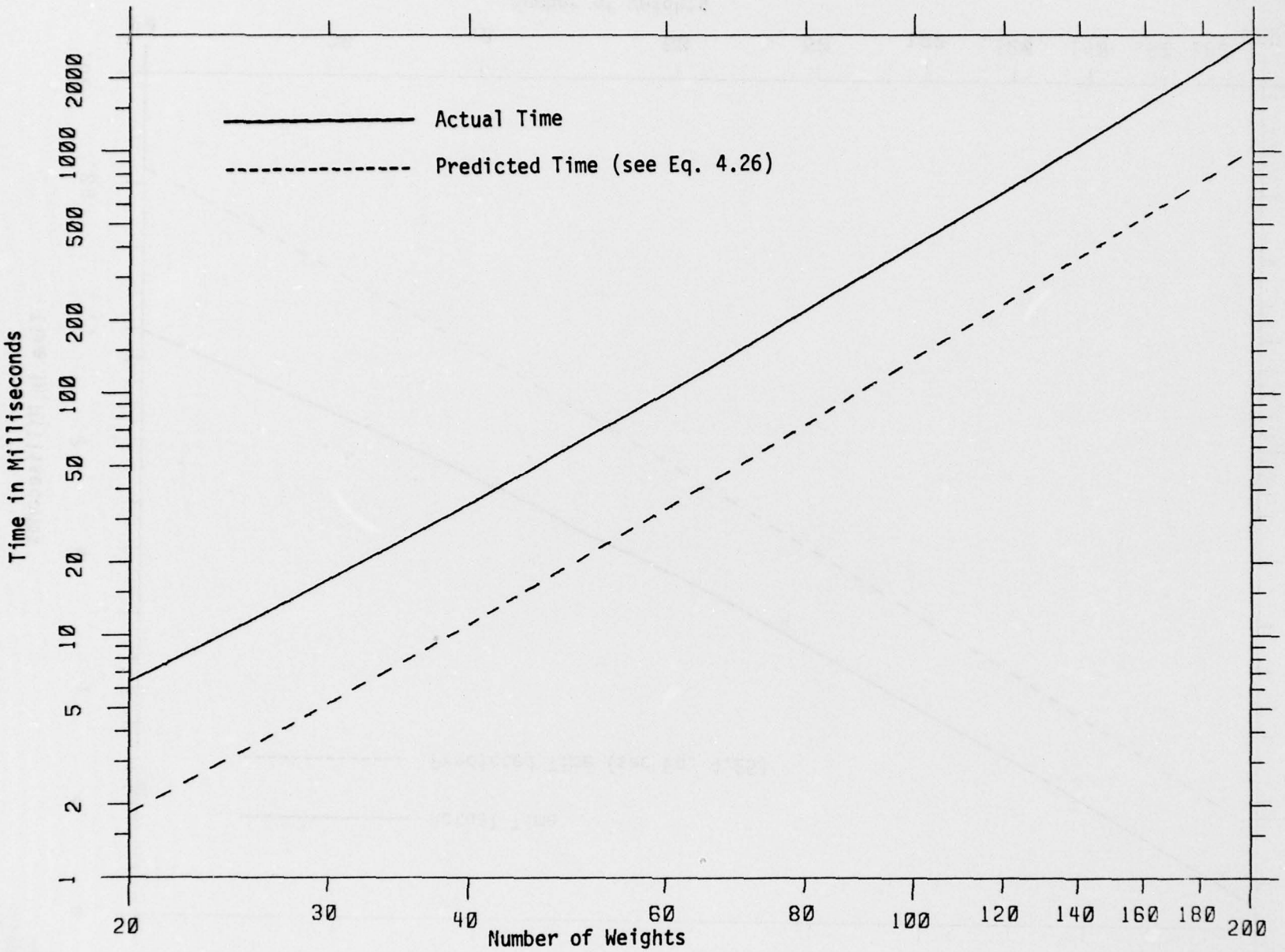



Figure 4.37 CRAY-1 Total Times to Process 2N Samples

weights, but the predicted times are consistently below the actual times, showing that the overhead ignored in the prediction contributes significantly and has about the same rate of growth as the prediction equation. One interesting difference between the STAR plots and the CRAY plots is that, while the actual times and predicted times tend to converge with the STAR, the CRAY times may either converge or diverge very slightly, indicating that there is more overhead ignored in our CRAY predictions than our STAR predictions. Some of this overhead may result from the extra startups needed to handle vectors whose length is greater than 64.

Table 4.16 contains the logarithmic timing predictions for the CRAY, obtained from doing linear least-squares fits to the logarithms of the times as functions of the logarithm of the numbers of weights. Here, as with the STAR and CDC 7600, the exponents (B values) are larger for the least-squares fits using the data satisfying $N > 90$ than for $N \leq 90$, but both are less than the exponents of the highest-order terms of the prediction equations (Eqs. 4.22 through 4.26): 3 for decomposition and total time, and 2 for both back substitutions and updating the matrix.

Table 4.16 Logarithmic Timing Predictions for the CRAY-1

| | All Data | | Data for Which N ≤ 90 | | Data for Which N > 90 | |
|---|---|---|---|---|---|---|
| | A | B | A | B | A | B |
| Decomposition | .000298 | 2.91 | .000353 | 2.86 | .000227 | 2.96 |
| First Back Substitution | .001200 | 1.50 | .002048 | 1.35 | .000436 | 1.70 |
| Second Back Substitution | .000765 | 1.92 | .000883 | 1.89 | .000610 | 1.97 |
| Update the Sample Covariance Matrix | .000800 | 1.57 | .001292 | 1.44 | .000322 | 1.75 |
| Total Time to Process 2N Samples | .001765 | 2.69 | .002718 | 2.57 | .000824 | 2.85 |

(Labels "A" and "B" are the constants in the linear least-squares equation

$$\log (\text{Time}) = B \log N + \log A \text{ or } \text{Time} = AN^B$$

where N is the number of weights.)

### 4.3.4 Conclusions for Vector Pipeline Processors

From the above discussion we conclude that calculating the sample covariance matrix and, second, solving $MW = \overline{S}$ by Cholesky decomposition of M and two back substitutions (or an augmented decomposition followed by one back substitution) are the optimal algorithms, given our assumption about the system configuration.

We may also draw some conclusions about the optimal form of vector pipeline processor for performing these algorithms. Our conclusions are functions of the system configuration; in particular, whether augmentation is possible (i.e., whether the steering vectors are available simultaneously with the covariance matrix) and how many steering vectors there are.

First, we would require a fast (hardware) square root. LL* decomposition is linear in the number of square roots and LDL* is quadratic in the number of MOVE operations, and hence LL* is a potentially faster algorithm. The back substitutions require more scalar multiplications with LL* than LDL*, but this too is a linear disadvantage.

Componentwise augmentation is superior in startups to a decomposition with separate back substitution if the system configuration permits. We must, however, also consider the number of steering vectors.

To fully take advantage of the difference in the number of startups as a function of storage scheme and number of steering vectors, we need a machine with sufficient core to store the entire matrix (actually only the upper half of the matrix has actual data and the lower half is for

spacing) and the ability to handle non-unity address increments. These characteristics allow us to access the data rowwise or columnwise, i.e., in any of the four storage schemes shown in Figure 4.20. Hence if K (the number of steering vectors) $\lesssim$ N/2, we may use the superior rowwise-vectorwise storage scheme for the first back substitution and the columnwise-vectorwise scheme for the second back substitution. If K $\gtrsim$ N/2, the rowwise-componentwise method is superior for both back substitutions.

## 4.4 IMPLEMENTATIONS TO DETERMINE WEIGHTS FOR A PARALLEL PROCESSOR

### 4.4.1 Introduction

The basic algorithms we will discuss will be forming the covariance matrix M and solving $MW = \overline{S}$ by direct methods. The reasons for not discussing iterative methods and loops are the same as discussed previously.

The basic means of comparison will be the number of processors required and operation counts. This analysis must be performed for each implementation, as parallel machines (both paper and actual machines) vary greatly from one to another. We will discuss the implementations for three existing machines: Goodyear Aerospace STARAN, PEPE, and ILLIAC IV. These and other parallel machines which have been announced, such as the Burroughs Scientific Processor (BSP) and the ICL Distributed Array Processor (ICL DAP) [Reddaway, 1973], are members of a class called Single Instruction Stream, Multiple Data Stream (SIMD) machines. This means that all active processors perform the same instruction at the same time. Another parallel organization is known as Multiple Instruction Stream, Multiple Data Stream (MIMD). These machines add another order of complexity to algorithm implementation and are generally used only for chaotic iteration or are programmed to act as SIMD machines [Conrad and Wallach, 1977].

With parallel machines, the number of processors can be extended to some limit; for example, the STARAN can have anywhere from 256 to 8192 processors in steps of 256. We will usually describe the optimal number of processors for a given implementation.

We will discuss the implementations based on machines with O(N) processors. In Section 3.2.7, we mentioned a technique requiring $O(N^5)$ processors, which

we view as impractical for a large number of weights. Appendix K details implementations for $O(N^2)$ processors. None of the machines mentioned above has a sufficient number of processors for this technique.

The algorithms whose implementations are to be examined are GE, GJ, LL*, and LDL*. The operations counted will be similar to those for vector and scalar machines, except that processor enable and data transfer instructions will also be counted.

The operation counts given are not the total number of operations but the number of steps for each type of operation. For example, N multiplies performed in parallel would count as one operation.

Since the matrix is Hermitian, only half of the matrix needs to be stored. This fact is very important in sequential and vector processors, but on a parallel machine it takes on a different aspect. We will assume that all parallel processors contain the same amount of memory. This assumption is valid on the STARAN, PEPE, ILLIAC IV, BSP and ICC DAP; and because each processor's memory size is identical, many such problems are reduced. If the machine has N processors, storing part of a row of M in a processor or storing the entire row demands the same amount of memory because at least one row is stored entirely. This does not imply that the fact M is Hermitian is not important. We must just examine its relevance from a different perspective.

The first machine we will discuss will be the PEPE because it can support a sufficient number of processors and memory to solve $MW = \overline{S}$ in the most straightforward manner. We feel that the PEPE's architecture is superior to those of other parallel machines we have examined to solve $MW = \overline{S}$ by direct methods. Appendix L details implementations for the PEPE. The

BSP is based on the PEPE, but it is too early to comment on its applicability.

The next machine to be discussed will be the ILLIAC IV, as implementations of our algorithms on the ILLIAC IV are somewhat akin to those on the PEPE. The STARAN will be discussed last.

In this section, we will also discuss the applicability of associative processing.

## 4.4.2 PEPE

### 4.4.2.1 Introduction to the PEPE

The PEPE (Parallel Element Processing Ensemble) is a parallel computer designed specifically for solving the radar site defense problem. This problem includes functions such as tracking targets, handling interceptors, and managing missile farms.

The PEPE is controlled by a CDC 7600 host computer and consists of three global control units which operate in parallel--the Correlation Control Unit (CCU), Arithmetic Control Unit (ACU), and Associative Output Control Unit (AOCU)--and an arbitrary number of Parallel Elements (PE), each of which contains three processors controlled by the three global control units and 2K 32-bit words of memory shared by the three processors. Figure 4.38 summarizes this structure. The current PEPE installation at the Ballistic Missile Defense Advanced Technology Center at Huntsville, Alabama has only 11 PE's, but emulator software is available for testing a 288-element configuration.

The CCU is a sequential processor with a 100-nsec cycle time. It is capable of supervisory instructions only and is used to perform input to the PE's via the CU's (Correlation Units), which it may operate in parallel. The CU's are capable only of indexing, data transfer, integer arithmetic, and logical functions.

The AOCU is also a sequential processor with a 100-nsec cycle time and is capable only of supervisory instructions to output data from the PE's via the AOU's (Associative Output Units), which it controls. The AOU's are like the CU's in that they operate in parallel and can perform only indexing, data transfer, integer arithmetic, and logical functions.

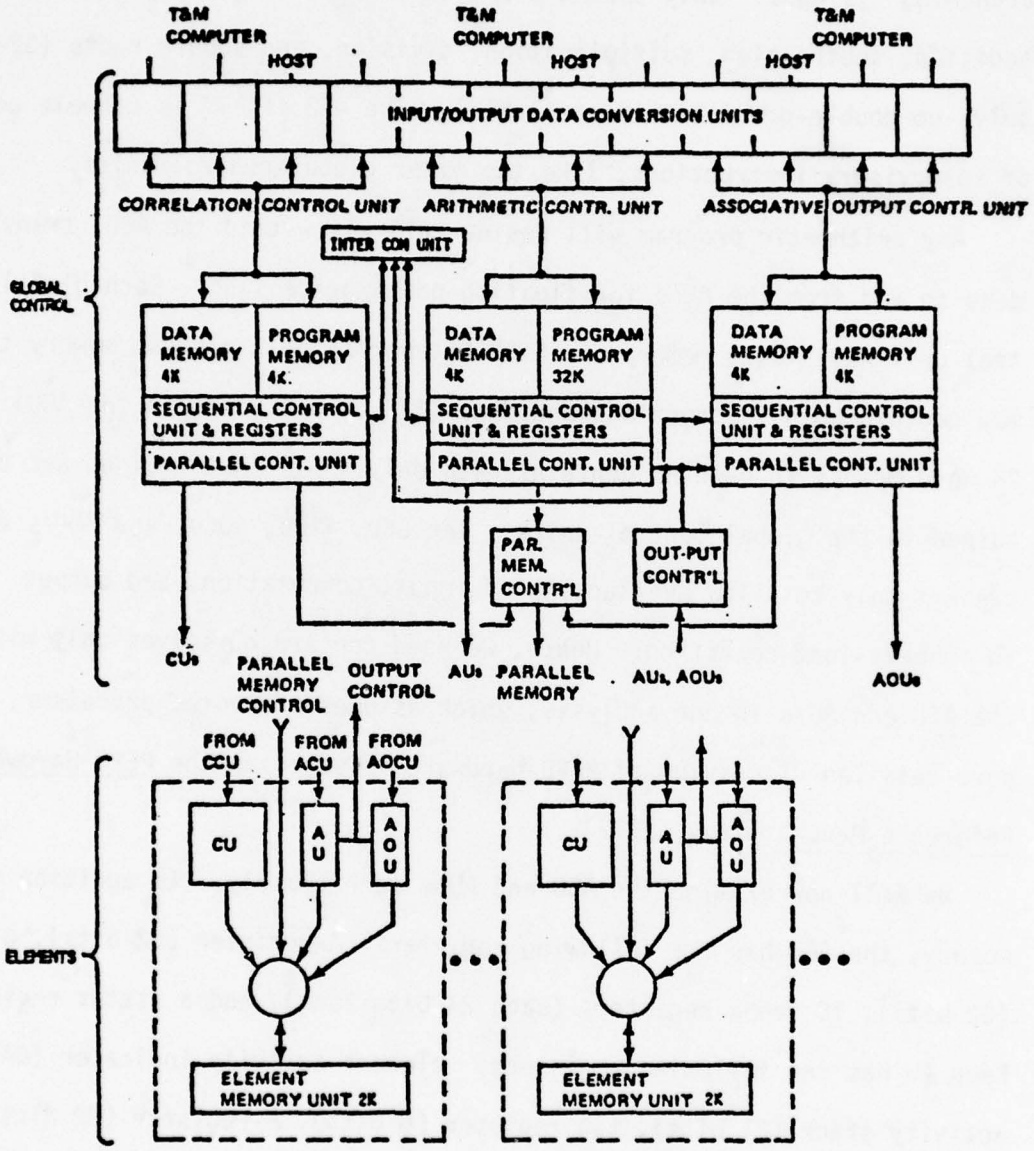



Figure 4.38 PEPE Architecture Block Diagram

The ACU controls the AU's (Arithmetic Units) where the actual "number crunching" is done. Only the AU's are capable of floating-point operations: addition, subtraction, multiplication, division, and square roots (32-bit only--no double-precision is available). The ACU itself is capable only of supervisory instructions, like the other global control units.

Any arithmetic program will reside entirely within the ACU, transferring data to and from the PE's for floating-point processing. Each global control unit has a data memory of 4K 32-bit words and a program memory that may contain both instructions and data (32K 32-bit words in the ACU). The 2K word memory in each PE contains data only. All instructions are contained in the global control units. The CCU, CU's, AOCU, and AOU's are present only to allow overlapping of input, computation, and output in a heavy-load condition. Hence, we need concern ourselves only with the ACU and AU's in our analysis, which assumes a devoted processor. For a more detailed discussion of PEPE hardware, please see the PEPE Hardware Reference Manual [Troy, 1977].

We will now examine the ACU and AU's more closely. In addition to its memory, the ACU has the following registers: A-register (32 bits), Q-register (32 bits), 16 index registers (each 24 bits long), and a status register. Each AU has the following registers: element activity indicator (EA) (1 bit), activity stack (21 bits), tag register (8 bits), A-register (32 bits), Q-register (32 bits), and several status registers. We can now discuss how the ACU executes instructions.

There are two instruction types, sequential and parallel (some instructions may fall under both categories, depending on operands, etc.). Sequential

instructions execute in the ACU and are of a supervisory nature, as discussed above. Parallel instructions are issued in the ACU, are passed to each AU simultaneously, and are performed in parallel in each PE whose element activity indicator (EA) is 1. There is a large repertoire of instructions for setting the EA. Setting a parallel element's EA to 1 is known as "enabling" that parallel element. The instructions available to enable and disable PE's include arithmetic tests on the values in the AU A-registers (.EQ., .NE., .GT., .LE. an operand, etc.), pushing and popping the activity stack that contains a record of past activity states, (pushing the stack moves each element in the stack down one and stores the current EA on top, and popping the stack replaces the EA with the top element of the stack and moves all the other elements up one), and testing the tag register (e.g., enable all processors the third bit of whose tag register is 1). All these instructions require different numbers of cycles, depending on their complexity.

For example, suppose the ACU issues an instruction to multiply the floating-point quantity x in location X of each active PE by the floating-point quantity y in location Y and store the result z in location Z. Each AU whose EA was set would simultaneously load A-register with x, multiply by y, and store the most significant bits of the result z in Z. The instruction would take the same amount of time, independent of the number of PE's enabled. The other floating-point operations would operate similarly.

The last hardware consideration to be discussed is the way data is passed between the ACU and the AU's. All communication is done via the A-registers. When passing data from the ACU to the AU's, the value in the sequential

A-register (ACU A-register) is simultaneously passed to the parallel A-registers (AU A-registers) of the AU's of all enabled PE's. To pass data in the other direction, only one PE may be enabled, in which case the value in that single parallel A-register may be passed to the sequential A-register. Using these two instructions together is the only way to pass data between different PE's--from the parallel A-register of the first PE to the sequential A-register to the parallel A-register of the second PE. To transfer large amounts of data between different PE's, there are several possible algorithms and implementations; choosing the fastest depends on the relative times it takes to store and load the A-registers, enable processors, and transfer data between A-registers, as discussed below. In any case, it is to an implementation's disadvantage to require a lot of inter-element data transfer, since the hardware is not designed to handle that function efficiently.

To perform an operation count analysis, we must have several primitive operations, as was done with the other architectures considered in other sections. All operations are assumed to take place in enabled PE's only, unless specified otherwise. Table 4.17 summarizes the operations, their abbreviations, descriptions, and timings as a multiple of the clock period (100 nsec). The timing of the ENABLE operation will depend on the complexity of the logical expression that must be evaluated in each PE to determine which ones will be enabled. The value given is the value for the most complicated one used (which is still a good approximation for the simple one-PE enables).

Since $t_{op}$ (*) $\leq 3 \cdot t_{op}$ ($\pm$) on the PEPE, complex multiplications (of numbers in the same PE) will be converted into 4 real multiplications and 2 real additions. Multiplying a complex number by a real or pure imaginary number, complex addition, complex sequential-to-parallel data transfer, complex

parallel-to-sequential data transfer and complex parallel data movement are all equivalent to 2 of their real counterparts. Only real operation counts will be given in all tables of operation counts.

Table 4.17 Operation Timings for the PEPE

| Operation | Description | Timings (multiple of 100-nsec clock period) |
|---|---|---|
| * | multiplication | 19 |
| ± | addition/subtraction | 8 |
| / | reciprocation | 38 |
| EN | parallel element enable | 8 |
| $\sqrt{}$ | square root | 32 |
| PM | parallel data movement (from one location in PE memory to another) | 3 |
| S-P | sequential-to-parallel data transfer | 4 |
| P-S | parallel-to-sequential data transfer (only one PE may be active) | 5 |

### 4.4.2.2 Implementations of Algorithms to Solve for Adaptive Weights on the PEPE

#### Introduction

As in preceding sections, the algorithms and implementations we discuss will be functions of the number of weights (N), the number of steering vectors (K), and the number of samples ($N_s$). We will analyze GE, GJ, LDL*, LL*, and their variations for their inherent parallelism, assuming O(N) parallel elements (at most N+K) are available. These algorithms consist mainly of row operations, making them well suited for implementation on a PEPE. But before discussing these methods, we must address the problems of storage schemes and transposing matrices.

There are three factors to consider when choosing a storage scheme. First, the scheme chosen should minimize the number of inter-element data transfers required. If two sets of values are always used in the same numeric expressions, such as the real and imaginary parts of each matrix element, or the elements of each column of the matrix, they should be stored in the same element. Second, as much parallelism should be used as possible (i.e., as many processors as possible should be enabled during each parallel operation). Hence, if the same operations are to be performed on different sets of data--the columns of the matrix, for example--those sets should be stored in separate elements. Third, the number of processor enables should be minimized. Thus, an implementation of an algorithm, such as GJ or GE, both of which perform sequences of row operations on a fixed set of columns, would require only one enable for each sequence of row operations; whereas, LDL* or LL*, both of which perform row operations on decreasing sets of columns, would require an enable for each set. What was LDL*'s and LL*'s advantage in a sequential or pipeline architecture is

now a disadvantage on the PEPE, since it performs a row operation in a constant time, independent of how many columns (elements) are involved (enabled).

The possible different storage schemes are summarized in Figs. 4.39a through 4.39d. Data items located in the same column are stored in the same parallel element, and data items located in the same row are stored in corresponding locations of different PE's. Figure 4.39a shows how the covariance matrix M is stored. Once factored, the factor matrix L may be stored in either of the two orientations shown in Fig. 4.39b. These orientations are called rowwise and columnwise, as an analog to the vector pipeline case. The steering vectors may be stored either vectorwise or componentwise, as shown in Fig. 4.39c. Finally, the abbreviations for the different possible combinations of storage schemes are shown in Fig. 4.39d.

So far, nothing has been said about the relative locations of the steering vectors and the sample covariance matrix. They may either be stored in separate processors, the matrix M occupying 2N locations of the first N PE's and the steering vectors occupying the next N (if stored vectorwise) or K (if stored componentwise) PE's, or they may both be stored in the first N PE's, in different locations. This second scheme is better (unless the implementation requires the first scheme) because it is far cheaper to make the memory in each PE a little larger than to make more PE's. On the PEPE, the memory in each PE is 2048 words, more than large enough to accommodate the second scheme for N and K up to 200. The only

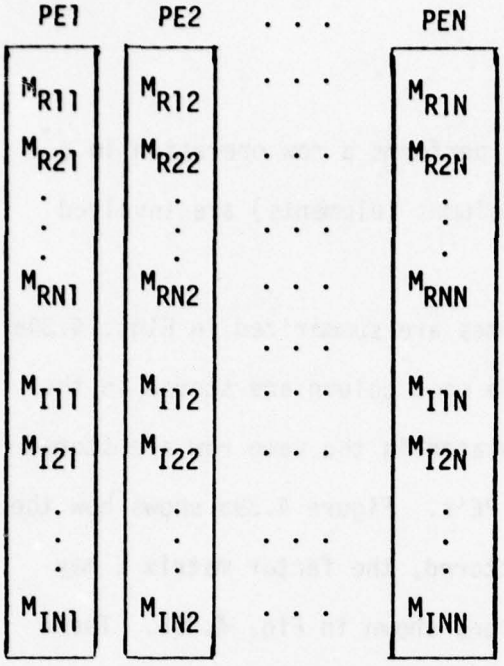



Figure 4.39a Storage Scheme for the Sample Covariance Matrix on the PEPE

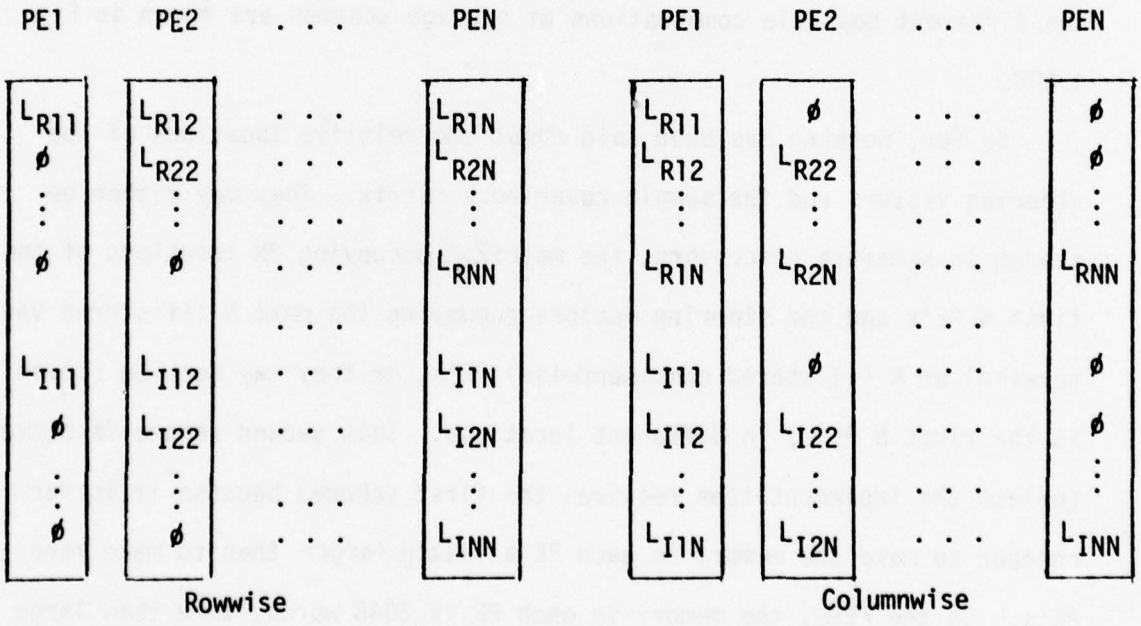



Figure 4.39b Storage Schemes of the Factored Sample Covariance Matrix on the PEPE

(ø denotes an arbitrary value.)

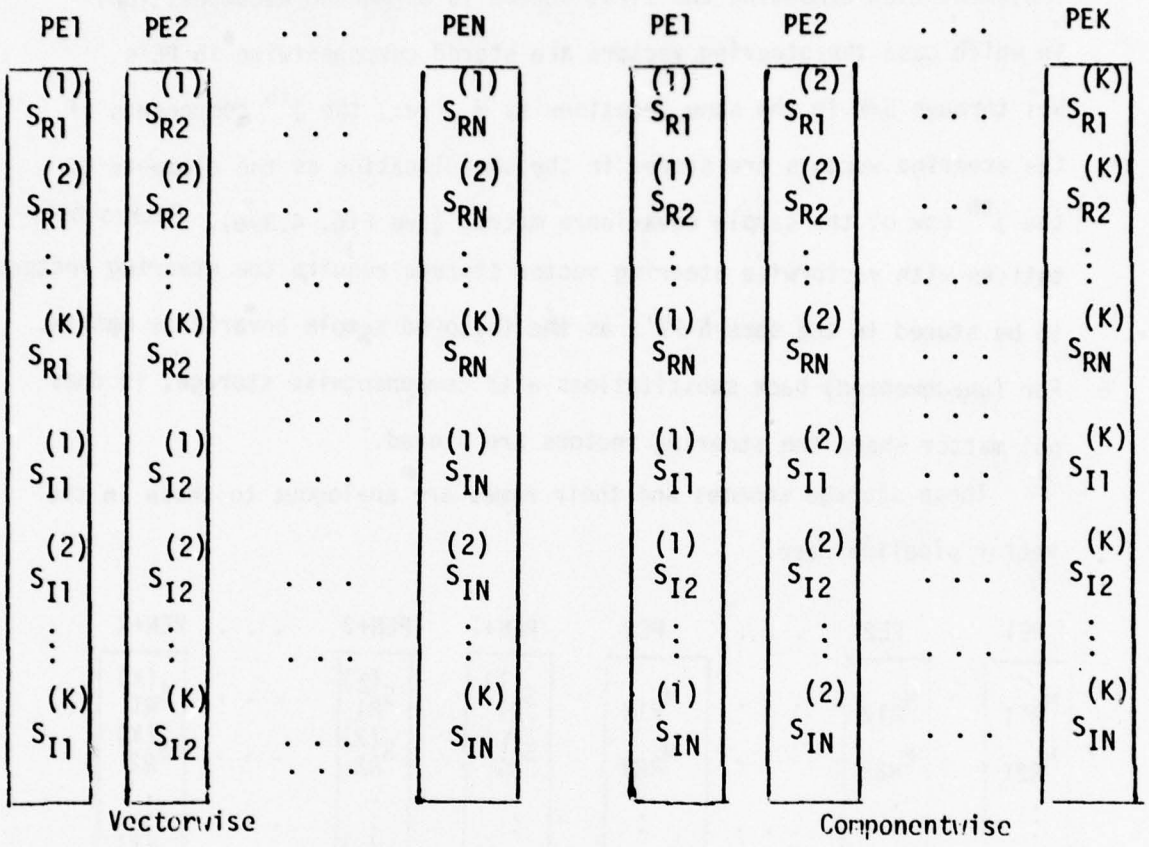



Figure 4.39c Storage Schemes for K Steering Vectors on the PEPE

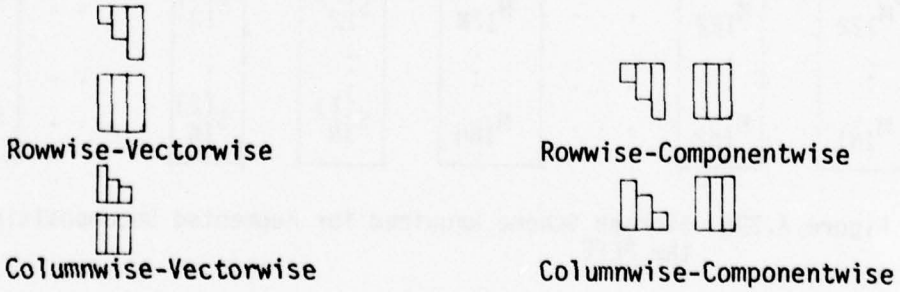



Figure 4.39d Abbreviations for Different Combinations of Factored Sample Covariance Matrix and Steering Vector Storage Schemes on the PEPE

implementation demanding the first scheme is augmented decomposition, in which case the steering vectors are stored componentwise in PE's N+1 through N+K in the same locations as M, i.e., the $j^{th}$ components of the steering vectors are stored in the same location as the elements of the $j^{th}$ row of the sample covariance matrix (see Fig. 4.39e). Back substitutions with vectorwise steering vector storage require the steering vectors to be stored in the same N PE's as the factored sample covariance matrix. For (unaugmented) back substitutions with componentwise storage, it does not matter where the steering vectors are stored.

These storage schemes and their names are analogous to those in the vector pipeline case.

| PE1 | PE2 | . . . | PEN | PEN+1 | PEN+2 | . . . | PEN+K |
|---|---|---|---|---|---|---|---|
| $M_{R11}$ | $M_{R12}$ | . . . | $M_{R1N}$ | $S_{R1}^{(1)}$ | $S_{R1}^{(2)}$ | . . . | $S_{R1}^{(K)}$ |
| $M_{R21}$ | $M_{R22}$ | . . . | $M_{R2N}$ | $S_{R2}^{(1)}$ | $S_{R2}^{(2)}$ | . . . | $S_{R2}^{(K)}$ |
| ⋮ | ⋮ | . . . | ⋮ | ⋮ | ⋮ | . . . | ⋮ |
| $M_{RN1}$ | $M_{RN2}$ | . . . | $M_{RNN}$ | $S_{RN}^{(1)}$ | $S_{RN}^{(2)}$ | . . . | $S_{RN}^{(K)}$ |
| | | . . . | | | | . . . | |
| $M_{I11}$ | $M_{I12}$ | . . . | $M_{I1N}$ | $S_{I1}^{(1)}$ | $S_{I1}^{(2)}$ | . . . | $S_{I1}^{(K)}$ |
| $M_{I22}$ | $M_{I22}$ | . . . | $M_{I2N}$ | $S_{I2}^{(1)}$ | $S_{I2}^{(2)}$ | . . . | $S_{I2}^{(K)}$ |
| ⋮ | ⋮ | . . . | ⋮ | ⋮ | ⋮ | . . . | ⋮ |
| $M_{IN1}$ | $M_{IN2}$ | . . . | $M_{INN}$ | $S_{IN}^{(1)}$ | $S_{IN}^{(2)}$ | . . . | $S_{IN}^{(K)}$ |

Figure 4.39e Storage Scheme Required for Augmented Decomposition on the PEPE

It may be that one storage scheme is best for one part of an algorithm but not for another (as in the vector pipeline case). If it were possible to rearrange the data from the one scheme to the other quickly enough, it might be faster to do that than to use the same storage scheme throughout processing. The transpose implementation chosen will depend not only on

the relative times it takes to perform the operations EN, S-P, and P-S, but also on the sequential memory size in words, $M_s$, and the shape of the matrix to be transposed (whether it is triangular or rectangular, and its dimension). We will first perform the analysis for a general machine, and then for the PEPE specifically, using the times in Table 4.17.

There are two types of methods, one using sequential memory and one using only the sequential A-register. This latter method transfers the data, one item at a time, from the memory of element i to its parallel A-register to the sequential A-register to the parallel A-register of element j to the memory of element j. The former reads as many columns (or rows) of the original matrix as will fit in sequential memory and then writes them back to parallel memory as rows (or columns). In both reading and writing, however, the data is accessed columnwise; i.e., all the data to be read from an element is read at one time, requiring one enable per column. If enough sequential memory is available to contain an entire r x c rectangular matrix, only r+c enables would be required (where r is the number of rows and c is the number of columns) in contrast to $2 \cdot r \cdot c$ for the second method. If the sequential memory is too small to contain the entire matrix, the number of enables will increase but will still be less than or equal to $2 \cdot r \cdot c$. To compare these methods requires an assembly language-level analysis with lower level instructions than we chose for our more general model of this architecture. The decision is very machine-dependent, and including primitives such as register loading and storing would make this analysis far too narrow. For the PEPE, the first method, using sequential memory, is superior.

Assume now we want to transpose a rectangular block of r rows and c columns (i.e., it occupies r locations in each of c parallel elements) as in Fig. 4.40a. Let $n_r$ be the number of complete columns that the sequential

memory (size = $M_s$ words) can contain, i.e., $n_r = \lfloor M_s/r \rfloor$. $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x, and $\lceil x \rceil$ denotes the smallest integer greater than or equal to x.) Similarly, let $n_c = \lfloor M_s/c \rfloor$ = the number of complete rows the sequential memory can hold. (If $M_s$ is more than large enough to contain the entire matrix of $r \cdot c$ words, i.e., $M_s > r \cdot c$, set $M_s = r \cdot c$.) There are two methods for performing the transpose using sequential memory. Method 1, illustrated in Fig. 4.40b, is as follows:

1) read a block of $n_r$ columns (or however many are left) of length r columnwise from parallel memory into sequential memory, requiring $n_r$ enables (or fewer, if fewer than $n_r$ columns remain)
2) write a block of $n_r$ (or fewer) rows of length r columnwise (r columns of length $n_r$) from sequential memory to parallel memory, requiring r enables
3) repeat steps 1) and 2) $\lceil c/n_r \rceil$ times.

Method 1 requires $r \cdot c$ S-P's, $r \cdot c$ P-S's, and

$$EN_1 = c + r \cdot \lceil c/n_r \rceil = c + r \cdot \lceil c/\lfloor M_s/r \rfloor \rceil \qquad 4.27$$

enables. Method 2, illustrated in Fig. 4.40c, is as follows:

1) read a block of $n_c$ rows (or however many are left) of length c columnwise (c columns of length $n_c$) from parallel memory into sequential memory, requiring c enables
2) write a block of $n_c$ (or fewer) columns of length c columnwise from sequential memory to parallel memory, requiring $n_c$ enables (or fewer if fewer than $n_c$ columns remain)
3) repeat steps 1) and 2) $\lceil r/n_c \rceil$ times.

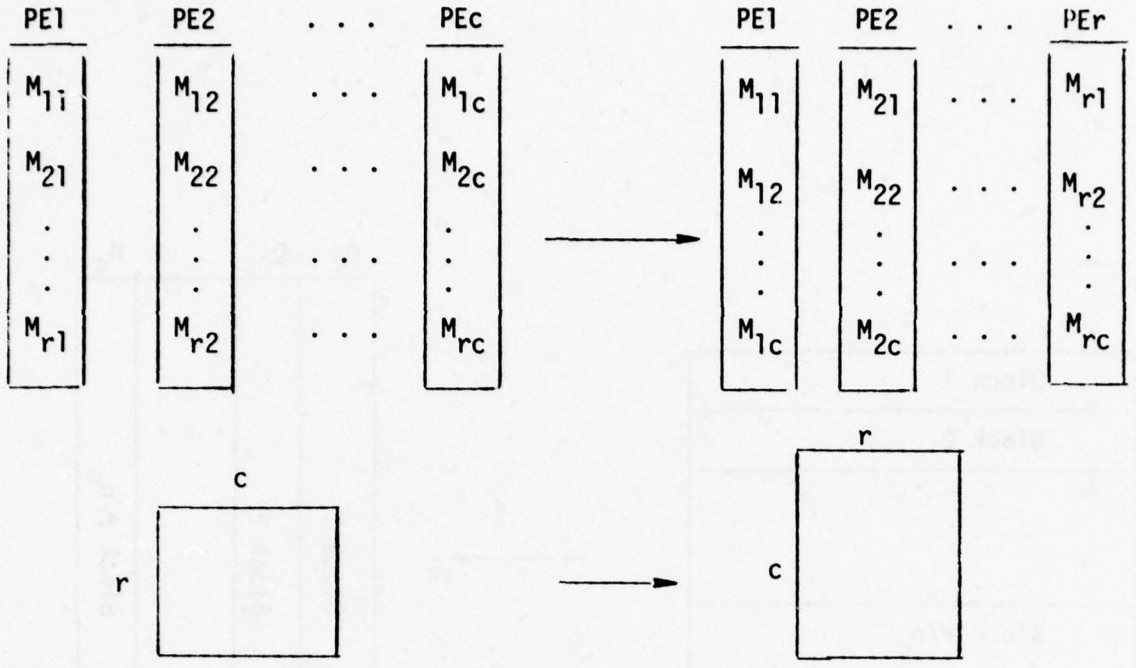



Figure 4.40a Transposing the r x c Matrix M on a PEPE

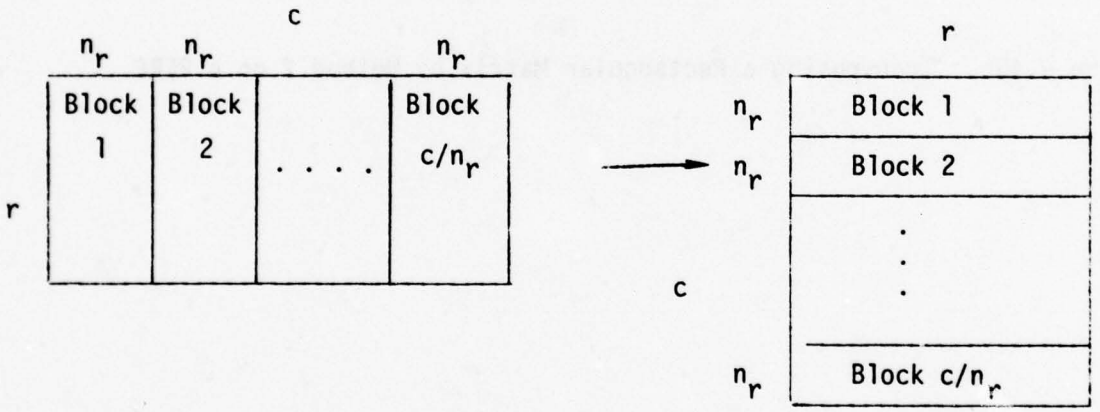



Figure 4.40b Transposing a Rectangular Matrix by Method 1 on a PEPE

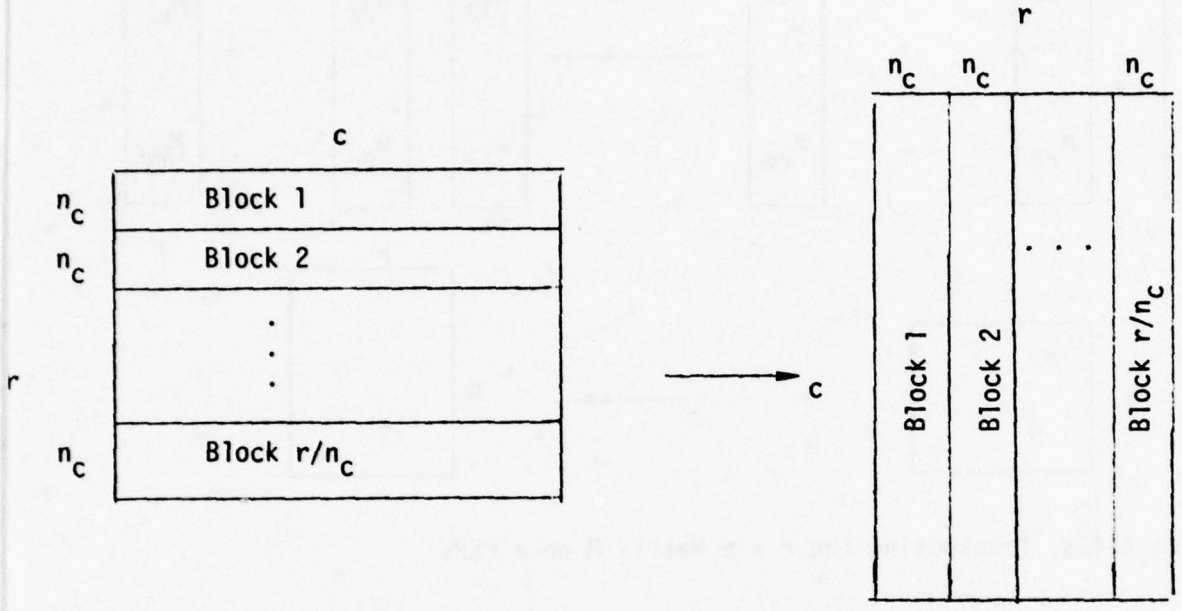



Figure 4.40c Transposing a Rectangular Matrix by Method 2 on a PEPE

Method 2 requires r·c S-P's, r·c P-S's, and

$$EN_2 = r + c \cdot \lceil r/n_c \rceil = r + c \cdot \lceil r/\lfloor M_s/c \rfloor \rceil \qquad 4.28$$

enables. Hence, Methods 1 and 2 differ only in the number of enables they require. In order to compare these numbers, we may approximate Eqs. 4.27 and 4.28 by

$$\tilde{EN}_1 = c + (c \cdot r^2)/M_s \qquad 4.29$$

and

$$\tilde{EN}_2 = r + (c^2 \cdot r)/M_s \quad . \qquad 4.30$$

It is straightforward to show that $\tilde{EN}_1 < \tilde{EN}_2$, and hence that Method 1 is superior to Method 2, if $c > r$. Similarly, Method 2 is superior if $r > c$. The two methods are approximately equivalent if $r = c$, but the exact equations, 4.27 and 4.28, must be used to determine the best method, particularly if r is close to c in value. In the best case, when $M_s = r \cdot c$, both methods require the minimum of r + c enables. Even if only one column or row fits in memory, the methods will require only c + r·c and r + r·c enables, respectively, far fewer than the 2·r·c enables of the "one-value-at-a-time" method.

The analysis of the complex case is somewhat different. Fig. 4.39c illustrates two storage schemes of a complex rectangular matrix that are the transposes of one another; note that the vectorwise scheme has 2K rows and N columns and its transpose has 2N rows and K columns, so that the dimensions of the one scheme are not simply the same dimensions reversed of the other scheme. Hence, Method 1 changes slightly to become Method 1' (where r=K, C=N, $n_r = \lfloor M_s/2r \rfloor$, $n_c$ = the largest even integer $\leq \lfloor M_s/c \rfloor = 2\cdot\lfloor \lfloor M_s/c \rfloor/2 \rfloor$, and $M_s \leq 2\cdot r\cdot c$):

1) read a block of $n_r$ (or fewer) columns of length 2r columnwise into sequential memory from parallel memory, requiring $n_r$ enables (or fewer)

2) write a block of $2n_r$ rows of length r (r columns of length $2n_r$) columnwise from sequential memory to parallel memory, storing real and imaginary parts in their appropriate places, requiring r enables

3) repeat steps 1) and 2) $\lceil c/n_r \rceil$ times.

Method 1' requires $2\cdot r\cdot c$ S-P's, $2\cdot r\cdot c$ P-S's, and

$$EN_1' = c + r\cdot\lceil c/n_r \rceil = c + r\cdot\lceil c/\lfloor M_s/2\cdot r \rfloor \rceil \qquad 4.31$$

enables. Method 2' is as follows:

1) read a block of $n_c$ rows (or fewer) of length c (c columns of length $n_c$, taking $n_c/2$ real parts and the corresponding $n_c/2$ imaginary parts, recalling that $n_c$ was chosen to be even) columnwise from parallel memory to sequential memory, requiring c enables

2) write a block of $n_c/2$ columns (or fewer) of length $2 \cdot c$ columnwise from sequential memory to parallel memory, requiring $n_c/2$ enables (or fewer)

3) repeat steps 1) and 2) $\lceil 2 \cdot r/n_c \rceil$ times.

Method 2' requires $2 \cdot r \cdot c$ S-P's, $2 \cdot r \cdot c$ P-S's and

$$EN_2' = r + c \cdot \lceil 2 \cdot r/n_c \rceil = r + c \cdot \lceil r/\lfloor \lfloor M_s/c \rfloor /2 \rfloor \rceil \qquad 4.32$$

enables. Using the same approximations as before,

$$\tilde{EN}_1' = c + (2 \cdot r^2 \cdot c)/M_s \qquad 4.33$$

and

$$\tilde{EN}_2' = r + (2 \cdot r \cdot c^2)/M_s \ , \qquad 4.34$$

and comparing them, we see that Method 1' is approximately superior to Method 2' if $c > r$; Method 2' is superior if $r > c$; and the two methods are approximately equivalent if $r = c$, although it is again advised to use the exact formulas, 4.31 and 4.32, to compare these methods. In the best case, when $M_s = 2 \cdot r \cdot c$, both methods again require the minimum number of enables, $r + c$.

We must also consider how to transpose the rowwise storage scheme of a triangular matrix to the columnwise storage scheme of Fig. 4.39b. There are again two methods available, analogous to the two for transposing rectangular matrices. Method 1, illustrated in Fig. 4.41a, reads complete columns from parallel to sequential memory and writes complete rows back

from sequential to parallel memory. Method 2, illustrated in Fig. 4.41b, reads complete rows from parallel to sequential memory and writes complete columns back from sequential to parallel memory. In each case, the number of columns (or rows) that will fit in sequential memory varies since their lengths vary, so it is difficult to give an analytic formula for the enables required. It is easy to show, however, that Method 2 will always require fewer enables than Method 1, and since both methods require $N\cdot(N+1)/2$ S-P's and P-S's ($N\cdot(N+1)$ for the complex case), we see Method 2 is always superior. The number of enables required in the best case, when sequential memory is large enough for the whole matrix ($M_s \geq N\cdot(N+1)/2$ in the real case, $M_s \geq N\cdot(N+1)$ in the complex case), is $2\cdot N$. This is the case on the PEPE (for which $M_s$ may be taken to be 4096, the size of the data memory in the ACU) for $N \leq 90$ in the real case and $N \leq 63$ in the complex case. The number of enables is approximately a quadratic function of N and is bounded above by $.0312\cdot N^2 + N$ for $28 \leq N \leq 200$ on the PEPE. The method for transposing the columnwise storage scheme to the rowwise storage scheme is just the reverse of Method 2 and has the same operation counts.

Forming the Sample Covariance Matrix on the PEPE

We will discuss ways to form the sample covariance matrix in the storage scheme shown in Fig. 4.39a. It turns out that this scheme is not only very convenient for the next part of the processing (solving $MW = \overline{S}$) but for the present matrix formation as well. Using this scheme, there are two implementations possible for forming M. We assume the sample vector is stored in sequential memory and must be moved into the PE's via S-P operations. Method 1

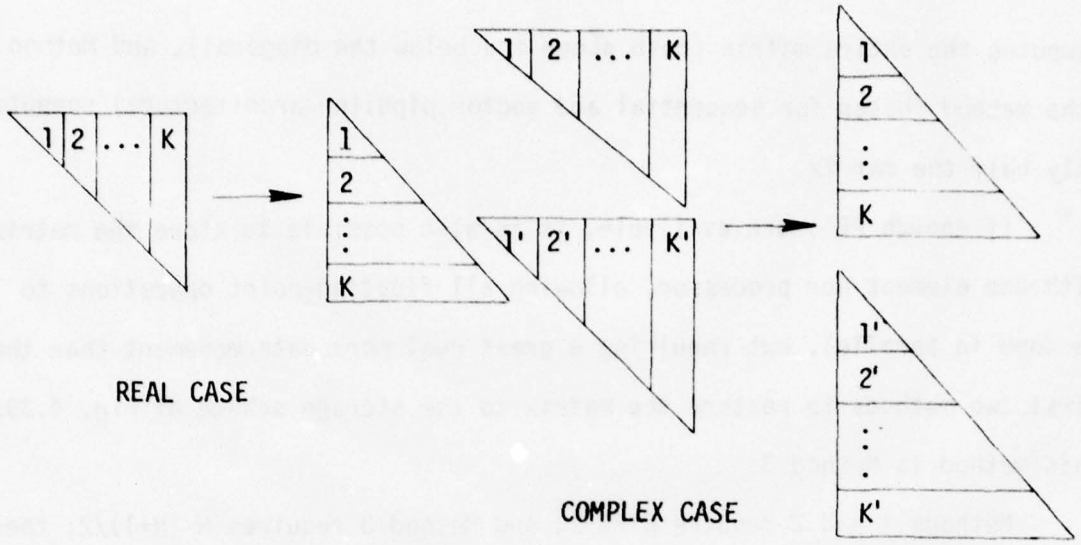



Figure 4.41a Method 1 for Transposing a Triangular Matrix on the PEPE (each numbered block is transposed from parallel to sequential to parallel memory in given order)

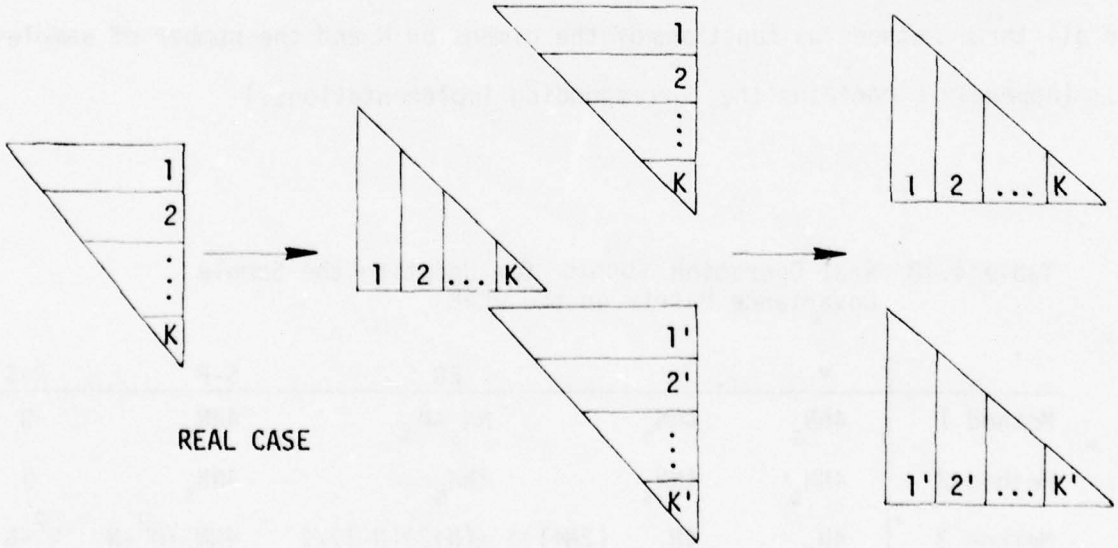



Figure 4.41b Method 2 for Transposing a Triangular Matrix on the PEPE (see comment for Fig. 4.41a)

computes the entire matrix (both above and below the diagonal), and Method 2 (the method chosen for sequential and vector pipeline architecture) computes only half the matrix.

If enough PE's are available, it is also possible to store the matrix with one element per processor, allowing all floating-point operations to be done in parallel, but requiring a great deal more data movement than the first two methods to restore the matrix to the storage scheme of Fig. 4.39a. This method is Method 3.

Methods 1 and 2 reqùire N PE's, and Method 3 requires $N\cdot(N+1)/2$; therefore, Method 3 is not practical for this architecture since we are assuming only O(N) processors are available, and the PEPE itself can support only 288 PE's, meaning Method 3 will fit only if $N \leq 23$. In addition, Method 3's operation counts depend on sequential memory size. The best case only, when $M_s > N\cdot(N-1)/2$, is shown in Table 4.18. Table 4.18 contains operation counts for all three methods as functions of the dimension N and the number of samples $N_s$. (Appendix L contains the corresponding implementations.)

Table 4.18 Real Operation Counts for Updating the Sample Covariance Matrix on the PEPE

| | * | ± | EN | S-P | P-S |
|---|---|---|---|---|---|
| Method 1 | $4NN_s$ | $4NN_s$ | $NN_s+N_s$ | $4NN_s$ | 0 |
| Method 2 | $4NN_s$ | $4NN_s$ | $2NN_s$ | $4NN_s$ | 0 |
| Method 3 | $4N_s$ | $4N_s$ | $(2N+1)N_s+(N+2)(N-1)/2$ | $4NN_s+N^2-N$ | $N^2-N$ |

We note, by examining Table 4.18, that Method 2, superior in a sequential or vector pipeline architecture, is no longer superior to Method 1, because the parallelism used makes the computation of an entire row of M as fast as the computation of the half to the right of the diagonal. In fact, to compute just the half to the right of the diagonal requires a new enable for each row, making Method 1 superior. Method 3 can be seen to be superior to both of the other methods, substituting enables and data movements for floating-point operations. It requires, however, too many PE's and too much sequential memory, as mentioned before. Hence, Method 1 is the best implementation for the PEPE.

Solving MW = $\bar{S}$ on the PEPE

As mentioned above, we chose to implement versions of GE, GJ, LDL*, and LL* that exploit their inherent parallelism. These four methods use full row operations (i.e., they consist of a series of operations that multiply complete rows of the matrix by constants and add multiples of a row to other rows) and are optimal among methods which use full row operations, as discussed in Section 3.2.1. Full row operations are clearly quite compatible with PEPE's architecture with O(N) processors, and since GE, GJ, LDL*, and LL* each requires $O(N^3)$ (sequential) floating-point operations, we expect them to require $O(N^3)/O(N) = O(N^2)$ (parallel) floating-point operations on the PEPE, and indeed, this is the case. Partitioning algorithms, such as Strassen's method (see Section 3.2.5.2), which operate on smaller and smaller subproblems and require much data movement, are clearly ill-suited for implementation on the PEPE.

Our approach for this analysis is as follows: perform operation counts (for the operations listed in Table 4.17) for each possible part of an implementation (different decompositions and back substitutions listed below), sum the operation counts for different combinations of parts of implementations (which form correct implementations), and finally, compare the total operation counts of the different complete implementations. These operation counts will be functions of N (the number of weights), K (the number of steering vectors), and $M_S$ (the sequential memory size, 4096 on the PEPE). The parallel memory size (2048 on the PEPE) is assumed to be large enough so as not to impose any restrictions on implementation choice (which is the case on the PEPE).

GJ is considered as a complete implementation by itself. Decompositions include GE, LDL*, and LL*, all of which can either be augmented or not augmented. Augmented decompositions solve LDT = $\overline{S}$ when M = LDL* (LT = $\overline{S}$ when M = LL*). The back substitutions, either first or second back substitutions, are used after LL* or LDL*-GE decompositions, and use one of the four storage schemes of Fig. 4.39d. The first back substitutions solve LDT = $\overline{S}$ when M = LDL* (LT = $\overline{S}$ when M = LL*), and the second back substitutions solve L*W = T when M = LDL* (L*W = T when M = LL*). In combining these parts of implementations to form complete ones, it may be necessary to transpose either the steering vectors (S) or the matrix (M) in order to have the necessary storage scheme. Since the numbers of EN operations for these transpositions are complicated functions of N, K, and $M_S$, we will adopt the following notation: $EN_S$ (K,N,$M_S$) will be the number of

enables needed to transpose a rectangular block of complex numbers stored in N columns and 2·K rows (see Fig. 4.39c) with a sequential memory size of $M_S$.

$EN_S(K,N,M_S)$ is given by Eq. 4.31 (with c = N and r = K) since $K \leq N$ (the number of independent steering vectors is necessarily less than the number of weights)

$$EN_S(K,N,M_S) = N+K*\lceil N/\lfloor M_S/(2*K)\rfloor\rceil \quad .$$

$EN_M(N,M_S)$ will be the number of enables required to transpose an N x N triangular matrix with a sequential memory of size $M_S$. The implementation determining $EN_M(N,M_S)$ is discussed in the first subsection of Section 4.4.2.2.

Forming the inverse matrix and multiplying it by the steering vector may be immediately ruled out as an efficient implementation because of the large amount of inter-element data transfer required to perform the needed dot-products.

Table 4.19a contains the operation counts for GJ; Table 4.19b contains the operation counts for unaugmented GE, LDL*, and LL*; Table 419c, for augmented GE, LDL*, and LL*; Table 4.19d, for the first back substitution; and Table 4.19e, for the second back substitution. (The corresponding implementations are located in Appendix L.)

Table 4.19a Real Operation Counts for GJ on the PEPE

| * | ± | / | EN | S-P | P-S |
|---|---|---|---|---|---|
| $4N^2-2N$ | $4N^2-4N$ | N | 2N | $2N^2-N$ | $2N^2-N$ |

Table 4.19b Real Operation Counts for Unaugmented Decomposition on the PEPE

| | * | ± | / | √ | Parallel Move | EN | S-P | P-S |
|---|---|---|---|---|---|---|---|---|
| GE | $2N^2-2$ | $2N^2-2N$ | N | 0 | 0 | 2N-1 | $N^2-1$ | $N^2$ |
| LDL* (optimized) | identical to GE | | | | | | | |
| LDL* (un-optimized) | $2N^2-2$ | $2N^2-2N$ | N | 0 | 2N-2 | $N^2+N-1$ | $N^2-1$ | $N^2$ |
| LL* (optimized) | $2N^2+2N-2$ | $2N^2-2N$ | N+1 | 1 | 0 | 3N+1 | $N^2+N-1$ | $N^2+N$ |
| LL* (un-optimized) | $2N^2-2$ | $2N^2-2N$ | N | N | 0 | $N^2+N-1$ | $N^2-1$ | $N^2$ |

Table 4.19c Real Operation Counts for Augmented Decomposition on the PEPE

| | * | ± | / | √ | Parallel Move | EN | S-P | P-S |
|---|---|---|---|---|---|---|---|---|
| GE | $2N^2$ | $2N^2-2N$ | N | 0 | 0 | 2N | $N^2$ | $N^2$ |
| LDL* (optimized) | identical to GE | | | | | | | |
| LDL* (un-optimized) | $2N^2$ | $2N^2-2N$ | N | 0 | 2N-2 | $N^2+N$ | $N^2$ | $N^2$ |
| LL* (optimized) | $2N^2+2N$ | $2N^2-2N$ | N+1 | 1 | 0 | 3N+2 | $N^2+N$ | $N^2+N$ |
| LL* (un-optimized) | $2N^2$ | $2N^2-2N$ | N | N | 0 | $N^2+N$ | $N^2$ | $N^2$ |

Table 4.19d Real Operation Counts for the First Back Substitution on the PEPE

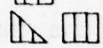
| | * | ± | EN | S-P | P-S |
|---|---|---|---|---|---|
| LDL*-GE | 4KN-2K | 4KN-4K | 2N-1 | 2KN-2K | 2KN-2K |
| LDL*-GE | $2N^2$ | $2N^2$-2N | 2N-2 | $N^2$ | $N^2$-N |
| LDL*-GE | transpose M, then use LDL*-GE implementation | | | | |
| LDL*-GE  | $2N^2$ | $2N^2$-2N | 2N-2 | $N^2$ | $N^2$-N |
| LL* | 6KN-4K | 4KN-4K | 2N-1 | 2KN+N-2K | 2KN-2K |
| LL*  | $2N^2$ | $2N^2$-2N | 2N-1 | $N^2$ | $N^2$-N |
| LL*  | transpose M, then use LL* implementation | | | | |
| LL* | $2N^2$ | $2N^2$-2N | 2N-1 | $N^2$ | $N^2$-N |

Table 4.19e Real Operation Counts for the Second Back Substitution on the PEPE

| | * | ± | EN | S-P | P-S |
|---|---|---|---|---|---|
| LDL*-GE | transpose M, then use LDL*-GE implementation | | | | |
| LDL*-GE | $2N^2$-2N | $2N^2$-2N | 2N-2 | $N^2$-N | $N^2$-N |
| LDL*-GE | 4KN-4K | 4KN-4K | 2N-2 | 2KN-2K | 2KN-2K |
| LDL*-GE | $2N^2$-2N | $2N^2$-2N | 2N-2 | $N^2$-N | $N^2$-N |
| LL*  | transpose M, then use LL* implementation | | | | |
| LL* | $2N^2$ | $2N^2$-2N | 2N-1 | $N^2$ | $N^2$-N |
| LL* | 6KN-4K | 4KN-4K | 2N-1 | 2KN-2K | 2KN-2K |
| LL*  | $2N^2$ | $2N^2$-2N | 2N-1 | $N^2$ | $N^2$-N |

We may draw several conclusions by examining these tables. From Tables 4.19b and 4.19c we see that LDL* and LL* are equal or inferior to GE in all operation counts. The versions of LDL* and LL* labelled "unoptimized" were written as straightforward conversions to the PEPE of algorithms used in previous sections, without attempting to minimize the number of EN's, S-P's, or P-S's. When optimized, they assumed the form of GE (with some extra code to multiply by the square roots of the diagonal elements for LL*). From Tables 4.19d and 4.19e, we see the operation counts for the LL* versions of the back substitutions are equal or inferior in all operation counts to the LDL*-GE versions (recall that GE and LDL* both provide the same decomposition of M, M=LDL*), so there is no advantage to LL* there either. Hence, LL* and LDL* were not considered any further. We note that any augmented decomposition (including GJ) requires N+K PE's for its most efficient implementation, whereas any other implementation requires only N.

By examining Tables 4.19d and 4.19e, we may note some striking analogs to Tables 4.8e and 4.8f, which contain the real operation counts for the first and second back substitutions on a vector pipeline processor. The number of startups (SU's) for addition and multiplication in Tables 4.8e and 4.8f are very close and often exactly equal to the number of additions and multiplications in Tables 4.19d and 4.19e. The storage schemes requiring the SUM operation on the vector pipeline processors correspond to the storage schemes requiring transposing the matrix on the PEPE. These analogs occur because having a sequence of data items lined up in core,

waiting to be fed into a pipe, is analogous to having those data items residing in the same location of adjacent PE's, waiting to be operated on in parallel. As long as no computation with different elements in the same sequence on a vector pipeline machine is required (no inter-element data transfer on the PEPE is required) this analogy will hold and the two architectures can be analyzed together. Unfortunately for decompositions, such computations (or inter-element data transfers on the PEPE) are required, and the optimal implementation on a vector pipeline machine, LDL* or LL*, turns out to be different from the optimal one on the PEPE, GE.

We are still left with a large number of different possible implementations, formed by combining either augmented GE with an LDL*-GE second back substitution and whatever transposes are needed to make storage schemes compatible, or unaugmented GE with an LDL*-GE first back substitution and an LDL*-GE second back substitution (with any required intervening transposes). It would be possible to transpose M before performing the first back substitution in the unaugmented case, but Table 4.19d indicates no advantage in the operation counts of the first back substitution could be achieved this way. All these possible implementations are listed in Tables 4.20a (augmented) and 4.20b (unaugmented). S-TRN denotes the process of transposing the matrix of steering vectors and M-TRN, the process of transposing M. When an implementation is labelled as identical to another, it is because it performs a transpose to the other implementation's storage scheme, which the other implementation lists under the

Table 4.20a Possible Implementations (Augmented) for Solving MW = $\bar{S}$ on the PEPE

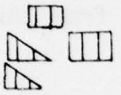
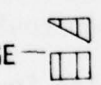
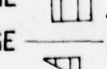
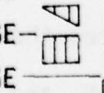
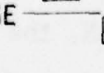
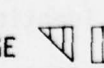
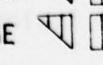
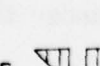
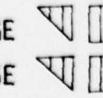
| IMPLEMENTATION NO. | DECOMPOSITION | S-TRN NEEDED? | M-TRN NEEDED? | SECOND BACK SUBSTITUTION STORAGE SCHEME | COMMENTS | TO BE CONSIDERED FURTHER? |
|---|---|---|---|---|---|---|
| 1 | GJ | — | — | — | | YES |
| 2 | GE | NO | NO | | | YES |
| 3 | GE | YES | NO | | identical to #5 | NO |
| 4 | GE | NO | YES |  | same operation count as #2 but requires M-TRN | NO |
| 5 | GE | YES | YES | | | YES |

Table 4.20b Possible Implementations (Unaugmented) for Solving MW = $\bar{S}$ on the PEPE

| IMPLEMENTATION NO. | DECOMPOSITION | FIRST BACK SUBSTITUTION STORAGE SCHEME | S-TRN NEEDED? | M-TRN NEEDED | SECOND BACK SUBSTITUTION STORAGE SCHEME | COMMENTS | TO BE CONSIDERED FURTHER? |
|---|---|---|---|---|---|---|---|
| 6 | GE |  | NO | NO | | identical to #7 | NO |
| 7 | GE |  | NO | YES | | | YES |
| 8 | GE |  | YES | NO | | | YES |
| 9 | GE |  | YES | YES | | same operation count as #8 but requires M-TRN | NO |
| 10 | GE |  | NO | NO | | | YES |
| 11 | GE |  | NO | YES | | same operation count as #10 but requires M-TRN | NO |
| 12 | GE |  | YES | NO | | identical to #13 | NO |
| 13 | GE |  | YES | YES | | | YES |

column "M-TRN Needed?". It is eliminated from further consideration but its mate is not. If an implementation is listed as having the same operation counts as a second implementation, except for requiring M-TRN, it is because both storage schemes (M transposed or not) result in the same operation counts; but since the first implementation has to perform the extra operations of an M-TRN, it is clearly inferior to the second implementation and eliminated from further consideration.

The complete operation counts for the remaining implementations are given in Tables 4.21a and 4.21b. These counts are obtained by adding up the appropriate values from Tables 4.19a through 4.19e and from the discussion of transposing matrices in the subsection "Introduction" of Section 4.4.2.2. Note that the number of enables is $EN_S(N,K,M_S)$ if the transpose is going from componentwise to vectorwise storage and $EN_S(K,N,M_S)$ if the transpose is going from vectorwise to componentwise storage. The number of enables required to transpose M is $EN_M(N-1,M_S)$ instead of $EN_M(N,M_S)$, because the diagonal elements of M (after factoring) are all known a priori to be one and need not be moved, reducing the problem's size to N-1 from N.

Among the augmented implementations, we note Implementation 2, GE without any transposes, is equal or inferior in all operation counts to Implementation 1, GJ, and hence may be eliminated from further consideration. Among the unaugmented implementations, #13 may be eliminated as inferior to #8.

The two remaining augmented implementations' operation counts for * and $\pm$ are dominated by the terms $4N^2$ and $2N^2+4KN$ for #1 and #5, respectively. These terms are equal when K = N/2, #5 being superior when $K < N/2$ and

Table 4.21a Total Real Operation Counts for Implementations for Solving $MW = \bar{S}$ on the PEPE

| IMPLEMENTATION NO. | AUG. OR UNAUG. | * | ± | / | EN | S-P | P-S |
|---|---|---|---|---|---|---|---|
| 1 | AUG. | $4N^2-2N$ | $4N^2-4N$ | N | $2N$ | $2N^2-N$ | $2N^2-N$ |
| 2 | AUG. | $4N^2-2N$ | $4N^2-4N$ | N | $4N-2$ | $2N^2-N$ | $2N^2-N$ |
| 5 | AUG. | $2N^2+4KN-4K$ | $2N^2+4KN-4K-2N$ | N | $4N-2+EN_S(N,K,M_S)+EN_M(N-1,M_S)$ | $2N^2+4KN-N-2K$ | $2N^2+4KN-N-2K$ |
| 7 | UNAUG. | $2N^2+8KN-6K-2$ | $2N^2+8KN-2N-8K$ | N | $6N-4+EN_M(N-1,M_S)$ | $2N^2+4KN-N-4K-1$ | $2N^2+4KN-N-4K$ |
| 8 | UNAUG. | $4N^2+4KN-2N-2K-2$ | $4N^2+4KN-4N-4K$ | N | $6N-4+EN_S(K,N,M_S)$ | $2N^2+4KN-N-2K-1$ | $2N^2+4KN-N-2K$ |
| 10 | UNAUG. | $6N^2-2N-2$ | $6N^2-6N$ | N | $6N-5$ | $3N^2-N-1$ | $3N^2-2N$ |
| 13 | UNAUG. | $4N^2+4KN-4K-2$ | $4N^2+4KN-4N-4K$ | N | $6N-5+EN_S(N,K,M_S)+EN_M(N-1,M_S)$ | $3N^2+4KN-N-2K-1$ | $3N^2+4KN-2N-2K$ |

Table 4.21b Total Real Operation Counts for Implementations for Solving $MW = \bar{S}$ on the PEPE When K=1

| IMPLEMENTATION NO. | AUG. OR UNAUG. | * | ± | / | EN | S-P | P-S |
|---|---|---|---|---|---|---|---|
| 1 | AUG. | $4N^2-2N$ | $4N^2-4N$ | N | $2N$ | $2N^2-N$ | $2N^2-N$ |
| 5 | AUG. | $2N^2+4N-4$ | $2N^2+2N-4$ | N | $5N-3+EN_M(N-1,M_S)$ | $2N^2+3N-2$ | $2N^2+3N-2$ |
| 7 | UNAUG. | $2N^2+8N-8$ | $2N^2+6N-8$ | N | $6N-4+EN_M(N-1,M_S)$ | $2N^2+3N-5$ | $2N^2+3N-4$ |
| 8 | UNAUG. | $4N^2+2N-4$ | $4N^2-4$ | N | $7N-3$ | $2N^2+3N-3$ | $2N^2+3N-2$ |
| 10 | UNAUG. | $6N^2-2N-2$ | $6N^2-6N$ | N | $6N-5$ | $3N^2-N-1$ | $3N^2-2N$ |

#1 being superior when K > N/2. The actual crossover point is a complicated function of N,K,$M_s$, and the different operation timings (t(*), t(±), t(EN), etc.) and is hence highly machine-dependent.

Among the unaugmented implementations, the * and ± operation counts are dominated by $2N^2+8KN$, $4N^2+4KN$, and $6N^2$ for #7, #8, and #10, respectively. Implementation 7's dominating term is superior to both others when K<N/2, all three are equal when K = N/2, and #10 is superior to both others when K>N/2. Implementation #8 may still be superior for a few values of K in the vicinity of N/2, depending again, in a complicated way, on N, K, $M_s$, and the different operation timings.

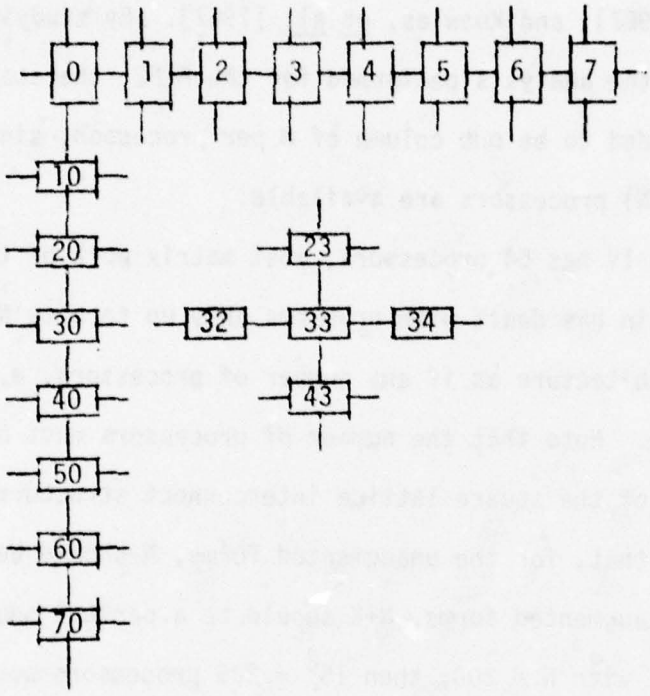
We note that the augmented implementations are superior to their unaugmented counterparts. When $K \leq N/2$, #5 is superior to #7 and #8; and when $K \geq N/2$, #1 is superior to #8 and #10. This situation is to be expected, since augmentation allows the first back substitution to be performed in parallel with the decomposition, costing no extra time except to divide the last row through by the last diagonal element, which costs 1 EN, 2 *'s, and 1 S-P.

In the case K = 1, we may be more specific. Under the weak assumption that $M_s \geq 2N$ (satisfied by the PEPE), we have $EN_s(K,N,M_s) = EN_s(N,K,M_s) = N+1$. The operation counts for the remaining implementations are given in Table 4.21b, where it is easy to pick #5 and #7 as the superior implementations, the augmented one (#5) being somewhat better than the unaugmented one (#7). If $M_s \geq 2N$, $EN_M$ will be bounded by N*(N+1)/2, which is more than made up for by the savings of approximately $2N^2-6N$ *'s and $2N^2-6N$ ±'s over Implementations 1 and 8.

Any more analysis will heavily depend on N, K, and the machine-dependent values of $M_s$ and the operation timings. By using Table 4.21a and the discussion of transposing matrices in Section 4.4.2.2, the optimal implementation can be chosen for any specific machine.

### 4.4.3 ILLIAC IV

The ILLIAC IV consists of 64 parallel processors connected in an 8 x 8 grid, with each processor able to communicate with its four neighbors, as illustrated below.




Each processor has 2048 words of 64 bits each and can perform integer and floating-point operations.

For a more detailed report of the ILLIAC IV, see ILLIAC IV: Systems Characteristics and Programming Manual [1972].

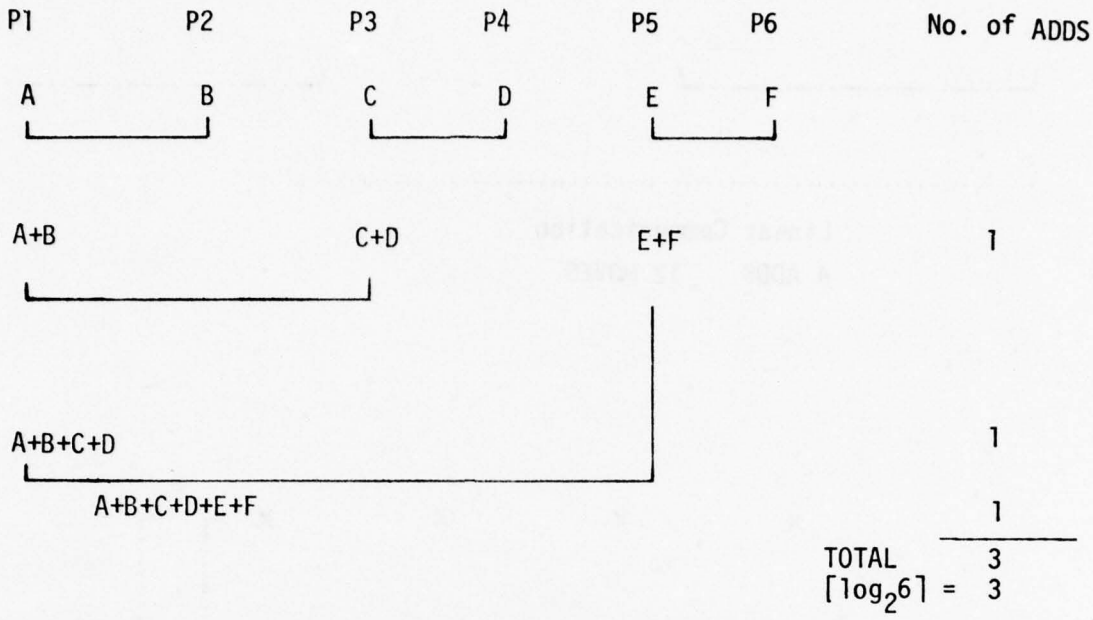
Because the ILLIAC IV has the capability to transfer data from one parallel element to all other elements, as on the PEPE, all of the analysis performed for the PEPE is applicable. The ILLIAC IV also has the ability of moving data between processors, which the PEPE does not

have. We will examine the use of this interconnect structure later in this section.

The literature contains quite a few papers on matrix methods on ILLIAC IV. The most germane are Pace [1972]; Edgar [1968]; Carr [1967]; Bernhard [1969]; Han [1967]; and Knowles, et al. [1967]. By studying these papers and using the analysis performed for the PEPE, the storage scheme for M was concluded to be one column of M per processor, since we are assuming only O(N) processors are available.

Because the ILLIAC IV has 64 processors, most matrix work of the type we are interested in has dealt with problems only up to size $N \leq 64$. We will examine the architecture as if any number of processors, e.g., 256, can be constructed. Note that the number of processors must be a perfect square because of the square lattice interconnect structure. This structure implies that, for the unaugmented forms, N should be a perfect square. For augmented forms, N+K should be a perfect square. Assuming unaugmented GE with N = 200, then $15^2 = 225$ processors would be required.

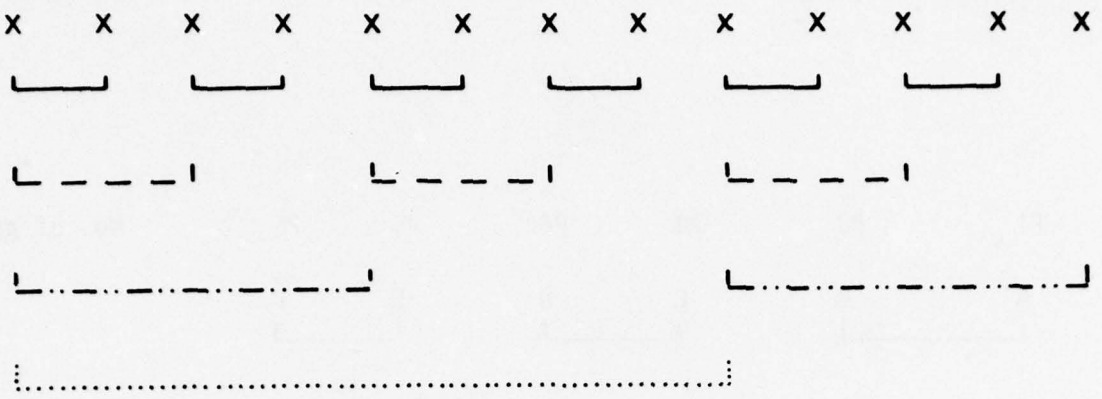
The ILLIAC IV interconnect structure is not required for the decomposition, as the implementation chosen for PEPE is optimal. For back substitutions, however, the interconnect structure may be useful. During some back substitution techniques, values must be summed across processors. With interprocessor communication, a sum of n items may be performed with $\lceil \log_2 n \rceil$ adds, as shown in the following illustration:



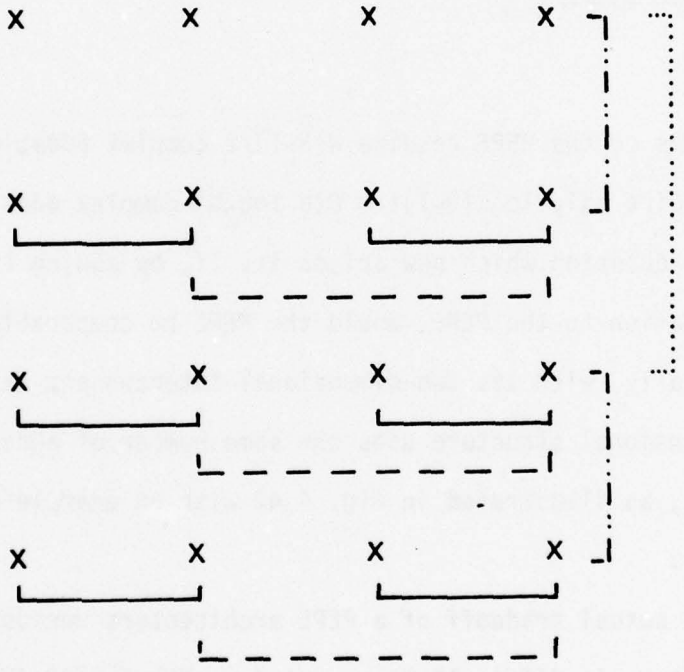

These sums on the PEPE require N(N-1)/2 complex adds; on the ILLIAC IV, they require only $\log_2(N-1)! = O(N \log_2 N)$ complex adds.

One question which now arises is, If, by adding linear interprocessor communication to the PEPE, would the PEPE be comparable with the ILLIAC IV, specifically, with its two-dimensional interconnect structure? The two-dimensional structure uses the same number of adds but less data movement, as illustrated in Fig. 4.42 with an example of adding 13 items together.

The actual tradeoff of a PEPE architecture versus an ILLIAC IV architecture is difficult to ascertain. The ILLIAC IV's interconnect structure is of only potential benefit during back substitution, which is a small operation compared with forming the covariance matrix and decompositions. Also, back substitution techniques exist which do not



Linear Communication
4 ADDS 12 MOVES



Two-Dimensional Interconnect (ILLIAC IV)
4 ADDS 6 MOVES

Figure 4.42 Parallel Adds of 13 Elements

LEGEND: _____ 1st ADD _ _ _ 2nd ADD

_._._. 3rd ADD ..... 4th ADD

require summing across processors. These techniques, which involve transposing, are developed in the PEPE section (Section 4.4.2).

The ILLIAC IV has been plagued with hardware problems since it was first brought up. These problems are not due to the architecture, but to the new devices and hardware invented for the machine [Falk, 1976]. There have been designs to make the ILLIAC IV fault-tolerant by switching on parallel elements when one fails. These designs have not yet been implemented. One problem with the ILLIAC's architecture is that the interconnect structure is optimized for a certain size matrix, e.g., 64 x 64. If a receiver, antenna element, or a set of analog-digital converters breaks so that the matrix decreases in size, the interconnect structure is no longer optimum [Bernhard, 1969]. The PEPE does not suffer from this loss of flexibility.

We feel that, while a machine with architecture similar to the ILLIAC IV can be used to solve for a set of weights, the PEPE architecture and algorithm implementation is superior because of hardware reliability and ease-of-programming considerations.

### 4.4.4 STARAN

#### 4.4.4.1 Introduction--Architecture of the STARAN

In a very general sense, the STARAN computer can be classified as a Content Addressable Parallel Processor (CAPP) [Foster, 1976], although the term does not entirely describe this unique machine. Within this chapter we will concern ourselves only with those portions of the STARAN that we found important and will criticize those portions with respect to performing the needed computations. The reader can obtain further details of the STARAN by referring to the STARAN Reference Manual [Goodyear, 1977].

For our own purposes, we viewed the STARAN as a Parallel Processor that possesses a Global Register (i.e., a register that could be transmitted to all processors) and a fair amount of interprocessor communication. For our particular problem of solving for the weights $MW = \overline{S}$, six items became apparent as important considerations: (1) No uses could be found for the associativity in the STARAN computer; (2) communication between processors is of questionable value, whereas a global register seemed almost a necessity; (3) the algorithm to be implemented is incredibly sensitive to the computer's architecture and to the dimension of the problem; (4) two machines are better than one; (5) the STARAN, due to its bit-serial nature, is slow in performing arithmetic computations even with the added benefit of its inherent parallelism; and (6) the STARAN computer is a very complex machine with no high-level language to ease development. Further examination of the STARAN architecture is necessary before commenting on these items.

The STARAN basic modular element is the Associative Processor Memory (AP), of which there can be a maximum of 32. All arrays are controlled by the

Associative Processor Control (AP control), which is serial in nature, obtaining its instructions and data from the Associative Processor Memory (AP memory). Each AP contains 256 Processor Elements (PE), which concurrently execute a single instruction stream. Each AP contains memory that is configured as 256 words, where a word is 256 bits wide for the current STARAN or is 9216 bits wide for the future STARAN-E. The width for memory access is 256 bits. The Common Register, C, is only 32 bits and has the capability of broadcasting information to all PE's (Global). Data transfer to and from an AP may occur either via the Common Register, or via the Parallel Input/Output Port (PIO), which is a 256-bit-wide I/O port; and in the case of the STARAN-E, via the crossbar network which looks like a 32-bit-wide block transfer path. An incredible amount of flexibilities in data access and configuration have been designed in the STARAN. In most of our analysis we treated the arrays in word mode, with each word subdivided into multiple 32-bit floating-point fields.

The following sections will reflect, with respect to the described architecture, the inherent problems and benefits one comes up with when solving for the weights in $MW = \overline{S}$ on the STARAN computer. The problem has been studied for a dimension up to 200 on the STARAN-B, -E, and a theoretical STARAN-type machine that utilizes the needed architectural characteristics in an expanded form.

One of our findings was that the associative aspects of the STARAN were not useful. Parallel computation of numerical values is very important in this problem. Associativity can be used in solving matrix equations by searching for the maximum diagonal element to perform pivoting. However, since the matrix M is positive definite, pivoting is not required.

There have been a few papers published discussing associativity and radar processing [Schaffer, 1976; Githens, 1970]. The use of associativity in these reports does not aid in the calculation of the weights. Other papers that deal with solving systems of equations using an associative processor (STARAN), Katz [1970], Gilmore [1975], Berra [1975], and Berra [1976], also do not utilize the associativity; they do heavily use the parallel capabilities of the STARAN.

#### 4.4.4.2 The Problem

Our problem is to find the weights in the equation $MW = \overline{S}$. This problem can be viewed in two separate steps, one being the formation of the covariance matrix and the other being the process of solving for the weights. Both of these steps are unique and will be treated separately.

#### Covariance Computation Method on a STARAN-Type Memory

To introduce the basic procedure for the covariance matrix computation, a generalized approach for a dimension of N will be examined and implemented on a STARAN-type memory where memory dimension is variable. Further analysis will follow with respect to the STARAN-B, and -E architectures.

Figure 4.43 represents an array memory map for processing of the covariance matrix. It is assumed that each field is 64 bits wide to hold the real and imaginary parts in a 32-bit floating-point representation; $X_i$'s represent the current input voltage vector and $m_{i,j}$'s represent the covariance matrix. The basic procedure is to input all 64*n bits of X into field 1 via a device like the PIO. Then replication via the Global Common Register of each $X_i$ into column $F_{2i}$ occurs. The complex conjugate of the X vector ($F_1 = F_1^*$) is then taken. The next step is to form the products, as follows: For the first element, Field 1 is multiplied by Field 2 and the results are placed into the spare field ($F_{sp}$), ($F_{sp} = F_1 \cdot F_{2i}$). This value is then averaged into the current average ($F_{2i+1} = F_{2i+1} + F_{sp}; F_{2i+1}/2$). This procedure is repeated for every column containing the replicated $X_i$'s and is noted in Figure 4.44.

In this general configuration, the memory configuration characteristics of the STARAN become apparent. The memory word-length dimension must be (64)(2n + 2), and the number of words needed is N. This structure will be

| | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | ... | $F_{2n}$ | $F_{2n+1}$ | $F_{sp}$ |
|---|---|---|---|---|---|---|---|---|---|
| $W_1$ | $X_1^*$ | $X_1$ | $m_{1,1}$ | $X_2$ | $m_{1,2}$ | ... | $X_n$ | $m_{1,n}$ | |
| $W_2$ | $X_2^*$ | $X_1$ | $m_{2,1}$ | $X_2$ | $m_{2,2}$ | | $X_n$ | $m_{2,n}$ | |
| $W_3$ | $X_3^*$ | $X_1$ | $m_{3,1}$ | $X_2$ | $m_{3,2}$ | | $X_n$ | $m_{3,n}$ | |
| . | . | | | | | | | | |
| . | . | | | | | | | | |
| . | . | | | | | | | | |
| $W_{n-1}$ | $X_{n-1}^*$ | $X_1$ | $m_{n-1,1}$ | $X_2$ | $m_{n-1,2}$ | | $X_n$ | $m_{n-1,n}$ | |
| $W_n$ | $X_n^*$ | $X_1$ | $m_{n,1}$ | $X_2$ | $m_{n,2}$ | | $X_n$ | $m_{n,n}$ | |

* = Complex Conjugate

$F_{sp}$ = Spare Field

Each field is 64 bits wide.

Figure 4.43 General Covariance Computation Data Structure for STARAN-Type Memory

(Unlimited Memory Dimension)
(Mode N Covariance Structure)

```
CLEAR MEMORY
FOR j=1 to 2n
    INPUT THE X VECTOR THROUGH A PORT TO F1
    FOR i=1 to n
        c= Wn,F1
        F2i=C
    END FOR
    F1 = F1*
    FOR i=1 to n
        Fsp= F1·F2i
        F2i+1 = F2i+1 + Fsp
    END FOR
END FOR
FOR i=1 to n
    F2i+1 = F2i+1/n
END FOR
```

Figure 4.44 Procedure for Covariance Computation
(Mode N Covariance Structure)

labeled "N mode covariance structure," relating the dimension of columns used. This basic scheme causes problems immediately for any real STARAN machine. Assume N = 200; then the word length must be 25,728 bits. We have only 9216 bits. Without change in the data structure, the maximum dimension that can be handled is 71, which could be accommodated by a single array. It is also noted that replication of each element of X into a full column (functionally equivalent to the outer product) is done efficiently by using the Common Register.

This structure (Fig. 4.43 ) can be improved by implementing more of the computation in parallel. By cutting the word length in half and doubling the number of words to be used, the product and averaging computation can be done for two columns' worth in the time it takes for 1. This structure and notated procedure are found in Figures 4.45 and 4.46, respectively. We will call the structure an N/2 - Mode Covariance Structure. Basically, by decreasing the number of columns used (decreasing the mode number) more parallelism occurs, thus reducing the time for calculations. Even further parallelism can occur by increasing the number of words to be used in a similar fashion, as stated above, until a data structure such as the one in Figure 4.47 is evolved. This is the limit case for speed reduction where the mode is 1. This last case capitalizes on the unlimited quantities of words to be used in order to reduce the computation count down to 1 product and 1 average.

There is one physical restriction: to inhibit this trend in massive parallelism. There exist only 32 x 256 available words on a maximum-sized STARAN. Using the procedure depicted in Figure 4.48 (Mode 4) and assuming a dimension of 200, 40,000 words would be needed; i.e., the maximum dimension

| | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ ... | $F_n$ | $F_{n+1}$ | $F_{sp}$ |
|---|---|---|---|---|---|---|---|---|
| $W_1$ | $X_1^*$ | $X_1$ | $m_{1,1}$ | $X_2$ | $m_{1,2}$ ... | $X_{n/2}$ | $m_{1,n/2}$ | |
| $W_2$ | $X_2^*$ | $X_1$ | $m_{2,1}$ | $X_2$ | $m_{2,2}$ | $X_{n/2}$ | $m_{2,n/2}$ | |
| $W_3$ | $X_3^*$ | $X_1$ | $m_{3,1}$ | $X_2$ | $m_{3,2}$ | $X_{n/2}$ | $m_{3,n/2}$ | |
| . | . | | | | | | | |
| . | . | | | | | | | |
| . | . | | | | | | | |
| $W_{n-1}$ | $X_{n-1}^*$ | $X_1$ | $m_{n-1,1}$ | $X_2$ | $m_{n-1,2}$ | $X_{n/2}$ | $m_{n-1,n/2}$ | |
| $W_n$ | $X_n^*$ | $X_1$ | $m_{n,1}$ | $X_2$ | $m_{n,2}$ | $X_{n/2}$ | $m_{n,n/2}$ | |
| $W_{n+1}$ | $X_1^*$ | $X_{n/2+1}$ | $m_{1,n/2+1}$ | $X_{n/2+2}$ | $m_{1,n/2+2}$ | $X_n$ | $m_{1,n}$ | |
| $W_{n+2}$ | $X_2^*$ | $X_{n/2+1}$ | $m_{2,n/2+1}$ | $X_{n/2+2}$ | $m_{2,n/2+2}$ | $X_n$ | $m_{2,n}$ | |
| . | . | | | | | | | |
| . | . | | | | | | | |
| . | . | | | | | | | |
| $W_{2n-1}$ | $X_{n-1}^*$ | $X_{n/2+1}$ | $m_{n-1,n/2+1}$ | $X_{n/2+2}$ | $m_{n-1,n/2+2}$ | $X_n$ | $m_{n-1,n}$ | |
| $W_{2n}$ | $X_n^*$ | $X_{n/2+1}$ | $m_{n,n/2+1}$ | $X_{n/2+2}$ | $m_{n,n/2+2}$ | $X_n$ | $m_{n,n}$ | |

* = Complex Conjugate
$F_{sp}$ = Spare Field
Each field is 64 bits wide.
Assume n is even.

Figure 4.45 Covariance Computation Data Structure for STARAN-Type Memory
(Mode N/2 Covariance Structure)

```
CLEAR MEMORY
FOR i=1 to 2n
    ENABLE WORDS 1 thru n
    INPUT X VECTOR TO F1
    ENABLE WORDS n=1 thru 2n
    INPUT X VECTOR TO F1
    FOR i=1 to n/2
        c=wi,F1
        ENABLE (W1-Wn)
        F2i=c
        c = W2i,F1
        ENABLE (Wn+1 - W2n)
        F2i=c
    END FOR
    F1=F1*
    FOR i=1 to n/2
        Fsp = F1*F2i
        F2i+1 = Fsp + F2i+1
    END FOR
END FOR
FOR i=1 to n/2
    F2i+1 = F2i+1/n
END FOR
```

Figure 4.46 Procedure for Covariance Computation
(Mode N/2 Covariance Structure)

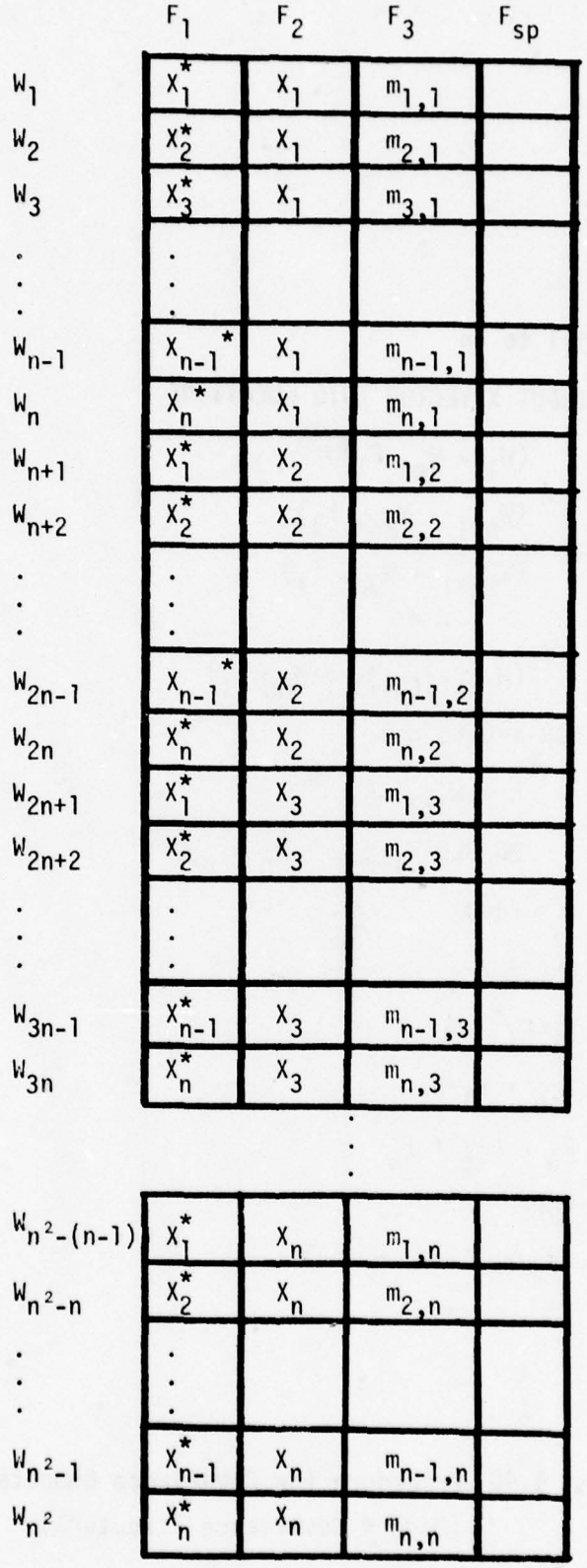



Figure 4.47 Covariance Computation Data Structure for STARAN-Type Memory

(Mode 4 Covariance Structure)

```
FOR j=1 to 2n
    INPUT X VECTOR INTO LOCATIONS
        (W1 - Wn, F1)
        (Wn+1 - W2n, F1)
        (W2n+1 - W3n, F1)
             :
        (W(n-1)(n)+1 - Wnn,F1)
    FOR i=0 to n-1
        c = Wi+1, F1
        ENABLE W1+n·i - Wn + n·i
        Fi=c
    END FOR
    F1=F1*
    Fsp = F1·F2
    F3 = Fsp + F3
END FOR
F3 = F3/n
```

Figure 4.48 Procedure for Covariance Computation
(Mode 4 Covariance Structure)

that could "fit" within this structure is $\sqrt{32 \times 256} \cong 90$. Working backwards, on a 32-array STARAN-B or -E, a 200-element covariance processor can be computed fastest by implementing it on a Mode 3 structure (Fig. 4.49). By using a STARAN-E (word length = 9216 bits) with only four arrays, the problem can be structured in Mode 50 (Fig. 4.50). This is about 4 times faster than the worst case, Mode 200, and it is about 16 times slower than the Mode 3 structure.

Each of the procedures notated in Figs. 4.48, 4.46, 4.44 generally proceeds in three steps. Step 1 can be characterized as input of the X vector; step 2, its outer-product data movement; and then step 3, its final calculation. For all modes, the outer-product operation time is constant. Already noted is the fact that the final calculation time can be reduced by decreasing the mode. The last thing to consider is whether input of the X vector to 67 segments (Mode 3) is substantially greater than just 4 inputs of the X vector as is the case for Mode 3. Using crossbar-type apparatus, the input-rate ratio (Mode 3 time : Mode 50 time) is about 17:1, where Mode 3 time is 1340 microseconds (μsec). This timing figure with respect to the faster STARAN-E times [Gilmore, 1975], where a complex multiple will be $4 \cdot 188 + 2 \cdot 118 = 988$ μsec, makes this data transfer insignificant compared to computation time, so that computation time must always be minimized.

The timing figures in Figure 4.51 are not accurate but do give relative timing ratios for the two modes calculated. Roughly, Mode 3 is 12 times faster than Mode 50. Since the STARAN calculation times are slow, reduction of all arithmetic operations is of prime importance for speed increase.

| | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ |
|---|---|---|---|---|---|---|---|
| $W_1$ | $X_1^*$ | $X_1$ | $m_{1,1}$ | $X_2$ | $m_{1,2}$ | $X_3$ | $m_{1,3}$ |
| $W_2$ | $X_2^*$ | $X_1$ | $m_{2,1}$ | $X_2$ | $m_{2,2}$ | $X_3$ | $m_{2,3}$ |
| $W_3$ | $X_3^*$ | $X_1$ | $m_{3,1}$ | $X_2$ | $m_{3,2}$ | $X_3$ | $m_{3,3}$ |
| ⋮ | ⋮ | | | | | | |
| $W_{n-1}$ | $X_{n-1}^*$ | $X_1$ | $m_{n-1,1}$ | $X_2$ | $m_{n-1,2}$ | $X_3$ | $m_{n-1,3}$ |
| $W_n$ | $X_n^*$ | $X_1$ | $m_{n,1}$ | $X_2$ | $m_{n,2}$ | $X_3$ | $m_{n,3}$ |
| $W_{n+1}$ | $X_1^*$ | $X_4$ | $m_{1,4}$ | $X_5$ | $m_{1,5}$ | $X_6$ | $m_{1,6}$ |
| $W_{n+2}$ | $X_2^*$ | $X_4$ | $m_{2,4}$ | $X_5$ | $m_{2,5}$ | $X_6$ | $m_{2,6}$ |
| ⋮ | | | | | | | |
| $W_{2n-1}$ | $X_{n-1}^*$ | $X_4$ | $m_{n-1,4}$ | $X_5$ | $m_{n-1,5}$ | $X_6$ | $m_{n-1,6}$ |
| $W_{2n}$ | $X_n^*$ | $X_4$ | $m_{n,4}$ | $X_5$ | $m_{n,5}$ | $X_6$ | $m_{n,6}$ |
| | | | ⋮ | | | | |
| $W_{65n}$ | $X_1^*$ | $X_{n-4}$ | $m_{1,n-4}$ | $X_{n-3}$ | $m_{1,n-3}$ | $X_{n-2}$ | $m_{1,n-2}$ |
| $W_{65n+1}$ | $X_2^*$ | $X_{n-4}$ | $m_{2,n-4}$ | $X_{n-3}$ | $m_{2,n-3}$ | $X_{n-2}$ | $m_{2,n-2}$ |
| | ⋮ | | | | | | |
| $W_{66n-1}$ | $X_{n-1}^*$ | $X_{n-4}$ | $m_{n-1,n-4}$ | $X_{n-3}$ | $m_{n-1,n-3}$ | $X_{n-2}$ | $m_{n-1,n-2}$ |
| $W_{66n}$ | $X_n^*$ | $X_{n-4}$ | $m_{n,n-4}$ | $X_{n-3}$ | $m_{n,n-3}$ | $X_{n-2}$ | $m_{n,n-22}$ |
| $W_{66n+1}$ | $X_1^*$ | $X_{n-1}$ | $m_{1,n-1}$ | $X_n$ | $m_{1,n}$ | NU | NU |
| $W_{66_{n+2}}$ | $X_2^*$ | $X_{n-1}$ | $m_{2,n-1}$ | $X_n$ | $m_{2,n}$ | NU | NU |
| | ⋮ | | | | | | |
| $W_{67n-1}$ | $X_{n-1}^*$ | $X_{n-1}$ | $m_{n-1,n-1}$ | $X_n$ | $m_{n-1,n}$ | NU | NU |
| $W_{67n}$ | $X_n^*$ | $X_{n-1}$ | $m_{n,n-1}$ | $X_n$ | $m_{n,n}$ | NU | NU |

NU = not used.

Figure 4.49 Covariance Computation Data Structure for STARAN-Type Memory

(Mode 3 Covariance Structure)

| | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | ... | | |
|---|---|---|---|---|---|---|---|---|
| $W_1$ | $X_1^*$ | $X_1$ | $m_{1,1}$ | $X_2$ | $m_{1,2}$ | ... | $X_{n/4}$ | $m_{1,n/4}$ |
| $W_2$ | $X_2^*$ | $X_1$ | $m_{2,1}$ | $X_2$ | $m_{2,2}$ | | $X_{n/4}$ | $m_{2,n/4}$ |
| ⋮ | | | | | | | | |
| $W_{n-1}$ | $X_{n-1}^*$ | $X_1$ | $m_{n-1,1}$ | $X_2$ | $m_{n-1,2}$ | | $X_{n/4}$ | $m_{n-1,n/4}$ |
| $W_n$ | $X_n^*$ | $X_1$ | $m_{n,1}$ | $X_2$ | $m_{n,2}$ | | $X_{n/4}$ | $m_{n,n/4}$ |
| $W_{n+1}$ | $X_1^*$ | $X_{n/4+1}$ | $m_{1,n/4+1}$ | $X_{n/4+2}$ | $m_{1,n/4+2}$ | | $X_{n/2}$ | $m_{1,n/2}$ |
| $W_{n+2}$ | $X_2^*$ | $X_{n/4+1}$ | $m_{2,n/4+1}$ | $X_{n/4+2}$ | $m_{2,n/4+2}$ | | $X_{n/2}$ | $m_{2,n/2}$ |
| ⋮ | | | | | | | | |
| $W_{2n-1}$ | $X_{n-1}^*$ | $X_{n/4+1}$ | $m_{n-1,n/4+1}$ | $X_{n/4+2}$ | $m_{n-1,n/4+2}$ | | $X_{n/2}$ | $m_{n-1,n/2}$ |
| $W_{2n}$ | $X_n^*$ | $X_{n/4+1}$ | $m_{n,n/4+1}$ | $X_{n/4+2}$ | $m_{n,n/4+2}$ | | $X_{n/2}$ | $m_{n,n/2}$ |
| $W_{2n+1}$ | $X_1^*$ | $X_{n/2+1}$ | $m_{1,n/2+1}$ | $X_{n/2+2}$ | $m_{1,n/2+2}$ | | $X_{3/4n}$ | $m_{1,3/4n}$ |
| $W_{2n+2}$ | $X_2^*$ | $X_{n/2+1}$ | $m_{2,n/2+1}$ | $X_{n/2+2}$ | $m_{2,n/2+2}$ | | $X_{3/4n}$ | $m_{2,3/4n}$ |
| ⋮ | | | | | | | | |
| $W_{3n-1}$ | $X_{n-1}^*$ | $X_{n/2+1}$ | $m_{n-1,n/2+1}$ | $X_{n/2+2}$ | $m_{n-1,n/2+2}$ | | $X_{3/4n}$ | $m_{n-1,3/4n}$ |
| $W_{3n}$ | $X_n^*$ | $X_{n/2+1}$ | $m_{n,n/2+1}$ | $X_{n/2+2}$ | $m_{n,n/2+2}$ | | $X_{3/4n}$ | $m_{n,3/4n}$ |
| $W_{3n+1}$ | $X_1^*$ | $X_{3/4n+1}$ | $m_{1,3/4n+1}$ | $X_{3/4n+2}$ | $m_{1,3/4n+2}$ | | $X_n$ | $m_{1,n}$ |
| $W_{3n+2}$ | $X_2^*$ | $X_{3/4n+1}$ | $m_{2,3/4n+1}$ | $X_{3/4n+2}$ | $m_{2,3/4n+2}$ | | $X_n$ | $m_{2,n}$ |
| ⋮ | | | | | | | | |
| $W_{4n-1}$ | $X_{n-1}^*$ | $X_{3/4n+1}$ | $m_{n-1,3/4n+1}$ | $X_{3/4n+2}$ | $m_{n-1,3/4n+2}$ | | $X_n$ | $m_{n-1,n}$ |
| $W_{4n}$ | $X_n^*$ | $X_{3/4n+1}$ | $m_{n,3/4n+1}$ | $X_{3/4n+2}$ | $m_{n,3/4n+2}$ | | $X_n$ | $m_{n,n}$ |

Figure 4.50 Covariance Computation Data Structure for STARAN

(Mode 50 Covariance Structure)

DATA TRANSFER = $T_{DT}$

OUTER PRODUCT = $T_{OP}$

CAL = $T_C$

| | Mode 3 | Mode 50 |
|---|---|---|
| $T_{DT}$ = | 200(67)(2)(.05) μs* | 200(4)(2)(.05) μs |
| $T_{OP}$ = | 200(1) μs | 200(1) μs |
| $T_C$ = | $3T_x + 3T_{\pm}$ | $50T_x + 50T_{\pm}$ |
| $T_x$ = | 4(188) +2(118) μs = 988 μs** | |
| $T_{\pm}$ = | 2(118) μs = 236 μs | |
| $T_{DIV}$ = | 333•2 (division by a real number) | |
| $T_D$ = | $3\ T_{DIV}$ = 1998 μs | $50\ T_{DIV}$ = 23300 μs |

COV CAL TIME = $2(200)\ [\ T_{DT} + T_{OP} + T_C] + T_D$

Mode 3 = 2(200) [1340 μs + 200 μs + 3675 μs] = 400[5215] + 1998 = 2.09 sec

Mode 50 = 2(200) [80 μs + 200 μs + 61250 μs] = 400[61530] + 23,300 = 24.64 sec

* Dimension = 200

** Each value is complex 32-bit floating-point; assume STARAN-E timing figures.

μs = μsec

Figure 4.51 Timing Figures for Covariance

The covariance calculation method described provides some insight into the architecture needed to solve such a problem. This problem is dimension- and architecturally sensitive; that is, given one implementation, an upward change in dimension could mean the adoption of a new algorithm. Architecturally it is sensitive in the sense that if the number of arrays (modes) to be utilized is low, then the lowest mode must be found that will fit. Basic features of the STARAN that are capitalized best are the global broadcasting capabilities of the Common Register and, of course, the parallelism afforded by having 256 processors per array.

Solution of $MW = \overline{S}$

In this section we will describe an implementation of the GJ algorithm only. Our reasoning is based on timing analysis of the STARAN and comparisons of the similarities and differences of the PEPE and STARAN.

Currently there is no hardware floating-point unit for the STARAN-B and none announced for the STARAN-E system. Berra [1976] has shown that, for N=200, a STARAN-B will invert the matrix in over 1 minute. The STARAN-E performs a multiply in 150 μsec. The CDC STAR-100C performs N multiplies in $(30 + \lceil \frac{N}{2} \rceil)$ 13.5 μsec. With the following equation, we will compute how many parallel multiplies the STARAN would need to compute to outperform a STAR-100C:

$$\frac{(30 + \lceil \frac{N}{2} \rceil)}{1000} \; 13.5 > .50$$

$$\lceil \frac{N}{2} \rceil > 11081.111$$

$$N > 22162.$$

The STARAN-E would need 22162 parallel processors before it would outperform a STAR-100C. Given these arguments, we feel that the STARAN is not a good machine to solve $MW = \overline{S}$.

The basic architecture for the STARAN may be good, except for slow floating-point math. The main features of the hardware are its associativity and interconnect structure. As we have stated earlier, the associative aspects of the STARAN are not applicable to the problem addressed in this report. The memory interconnect structure is also not required by the problem. As shown in the discussion on the ILLIAC IV, the interconnect structure may be of use for certain back substitution techniques.

As shown in Appendix L, the interconnect structure is not required for GJ.

By removing the associative and interconnect features of the STARAN, we obtain a machine very similar to the PEPE. The major difference is the size of processor memory. Since we know that GJ is optimal for O(N) processors on a PEPE, we will describe its implementation on a STARAN with an unlimited-memory-sized processor and with only 9K bits of processor memory, as in the STARAN-E. Basically, the technique is the same as that used for the PEPE, where the needed steering vector combinations are adjoined to the M matrix and diagonalization occurs on M with the operations affecting the S vectors. A data structure was developed to maximize the parallel nature of the STARAN, and algorithms were developed to handle the particular memory dimensions. *The main finding* was that algorithms are sensitive to dimension and architecture, if the entire $[M][S]^T$ is configured in the arrays. The storage schemes and programs for GJ are shown in Figs. 4.52, 4.53, and 4.54.

```
FOR  j=1 to n
    c = (W_j, F_j)
    FOR i=1 to j + k
        (W_j, F_i) = (W_j, F_i)/c
    END FOR
    FOR k = (1 to n) and k ≠ j
        c = -(W_k, F_j)
        FOR i=1 to j + k
            sp, F_i = (W_k, F_i)·c
            W_k, F_i = (W_k, F_i) + (sp, F_i)
        END FOR
    END FOR
END FOR
```

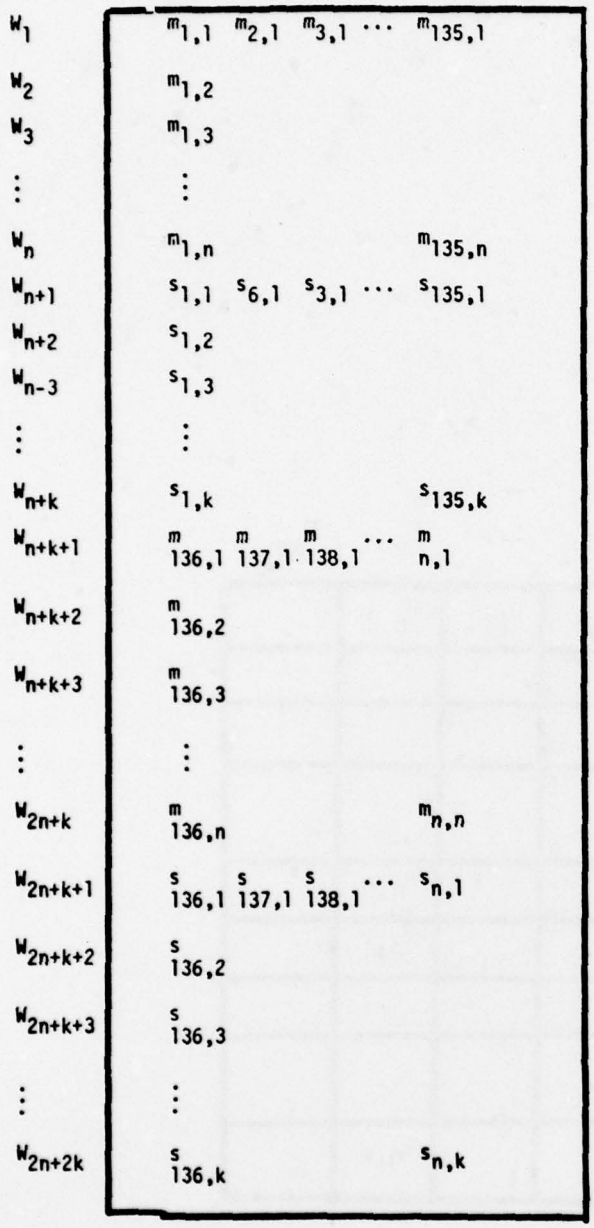
| | $F_1$ | $F_2$ | $F_3$ | ... | $F_n$ | $F_{n+1}$ | $F_{n+2}$ | ... | $F_{n+k}$ |
|---|---|---|---|---|---|---|---|---|---|
| $W_1$ | $m_{1,1}$ | $m_{1,2}$ | $m_{1,3}$ | ... | $m_{1,n}$ | $s_{1,1}$ | $s_{1,2}$ | ... | $s_{1,k}$ |
| $W_2$ | $m_{2,1}$ | | | | | | | | |
| $W_3$ | | | | | | | | | |
| ⋮ | | | | | | | | | |
| $W_n$ | $m_{n,1}$ | $m_{n,2}$ | ... | | $m_{n,n}$ | $s_{n,1}$ | | | $s_{n,k}$ |
| Sp | | | | | | | | | |

Figure 4.52 Algorithm for GJ with K Steering Vectors (Unlimited Word Size)

```
FOR j=1 to n
    c = W_j, F_j
    F_j = F_j/c
    FOR i = (1 to n) and i ≠ j
        C = -(W_j, F_i)
        F_sp = F_j • c
        F_i = F_i + F_sp
    END FOR
END FOR
```

| | $F_1$ | $F_2$ | $F_3$ | ... | $F_n$ | $F_{sp}$ |
|---|---|---|---|---|---|---|
| $W_1$ | $m_{1,1}$ | $m_{2,1}$ | $m_{3,1}$ | ... | $m_{n,1}$ | |
| $W_2$ | $m_{1,2}$ | | | | | |
| $W_3$ | $m_{1,3}$ | | | | | |
| ⋮ | ⋮ | | | | | |
| $W_n$ | $m_{1,n}$ | | | | $m_{n,n}$ | |
| $W_{n+1}$ | $s_{1,1}$ | $s_{2,1}$ | ... | | $s_{n,1}$ | |
| $W_{n+2}$ | $s_{1,2}$ | | | | | |
| ⋮ | ⋮ | | | | | |
| $W_{n+k}$ | $s_{1,k}$ | | | | $s_{n,k}$ | |

Figure 4.53 Algorithm for GJ with K Steering Vectors Transposed in the Array Memory
(Unlimited Word Size)




```
ENABLE (W1 - Wn+k)
FOR j=1 to 135
    c = Wj, Fj
    Fj = Fj/c
    TRANSFER (W1 - Wn+k), Fj to Wn+k+1-W2(n+k), Fp
    ENABLE (W1 - Wn+k)
    FOR i = (1 to 135) and I ≠ J
        c = - (Wj,Fi)
        Fsp = Fj•C
        Fi = Fi + Fsp
    END FOR
    ENABLE (Wn+k+1 - W2(n+k))
    FOR i=1 to n - 135
        c = -(Wn+k+j, Fi)
        Fsp = Fp•C
        Fi = Fi + Fsp
    END FOR
END FOR
ENABLE (Wn+k+1 - W2(n+k))
FOR j = 136 to n
    c = Wn+k+j, Fj, -135
    Fj - 135 = Fj-135/c
    TRANSFER (Wn+k+1 - W2(n+k),Fi-135)to(W1-Wn+k,Fp)
    FOR i = (1 to n-135) and (i ≠ j-135)
        c = -(Wn+k+j,Fi)
        Fsp = Fn+k+j•C
        Fi = Fi + Fsp
    END FOR
    FOR i = 1 to 135
        c = -(Wj, Fi)
        Fsp = Fp•C
        Fi = Fi + Fsp
    END FOR
END FOR
```

Figure 4.54 Program for GJ with K Steering Vectors Transposed in the Array with 9K Bit Word Size Where N < 270

## 4.5 HYPOTHETICAL SYSTEM

One conclusion of our analysis is that a computer optimized to solve $MW = \overline{S}$ will not necessarily be good to form M. We would propose a two-computer system--one computer forming M and the other computer solving for W. It is possible, as shown in Appendix I, for a computer to form M entirely with integer math; the formation can be made inexpensively and fast. The computer to solve $MW = \overline{S}$, however, does require floating-point math.

One thing that slowed down all of the implementations described is complex operations. If the computer were constructed with complex multiply instructions, many steps could be saved because the operation could proceed in parallel in two steps instead of in the usual six.

### 4.5.1 Calculation of M

We want to ensure that the estimate of M used reflects the state of the environment at the time we solve for the weights. By having a separate machine for this function, no samples need be lost during the time used to solve for the weights.

As shown in the mathematical technique section (Section 3.0), forming M from 2N samples requires $N^3$ multiplies, and Cholesky's dominant term is $N^3/6$. In sequential and vector processing, most of the time will be occupied with calculating M. By allowing overlap, we can calculate new weights without waiting for a new M to be calculated. On parallel machines, the difference in computational complexity between forming M and solving $MW = \overline{S}$ decreases since both algorithms are $O(N^2)$ for N processors and $O(N)$ for $N^2$ processors. With overlap, we will decrease the time between each weight calculation by a factor of 2.

What is the optimal architecture for the machine to calculate M? That is a difficult question, depending upon definition of "optimal," no. of degrees of freedom, speed, cost, etc.

In terms of speed, the optimal architecture to compute M will have $N^2$ (or N(N+1)/2, since the matrix is Hermitian) processors; that is, one processor for each element of M. Each processor will be capable of complex multiply and add operations with these operations pipelined to operate at the ADC rate on the radar receivers. Such a system is depicted in Appendix K. While this system is feasible today due to the simplicity of the processors and known fault-tolerant techniques to replace failing processors, the cost and power consumption for N = 200 and a sampling rate of 30 MHz will make it impractical.

In contrast to this fully parallel approach, we can use a dedicated machine such as a CDC STAR-100. There are problems with this approach, however. One problem is reliability. It would require at least two STARs to ensure a reliable system. At a price tag of $8 million per STAR, a considerable sum of money would be involved. The STAR takes 30 msec for each new value from the radar for N = 200. For 400 samples, 8 seconds would be required just to form M.

We will now propose a method between the two extremes discussed above. We propose a parallel processor approach with p processors (where $p \leq N$) which divide the work between them. From economic and reliability viewpoints, this approach is better than either of the preceding. A parallel processor requires spare processors in case of failure and not an entire new machine, so we will retain good reliability at low cost. Fast processors today are becoming inexpensive. The Floating Point Systems AP120B costs $30,000 and performs complex multiply and add in .667 msec. Ten of these machines working in parallel would cost about $300,000 and compute M at roughly the same rate as the $8-million STAR.

The AP120B, however, is much more complicated than needed for this task. We would recommend designing a particular parallel processor for this task.

An important consideration is the amount of memory per processor, as the STARAN showed how insufficient memory per processor raises the complexity of the system. Also, we would design the system so that any number of processors up to some limit can be used so that, for different radar installations with different numbers of weights, no redesign would be necessary. This flexibility also makes it easier to store spare parts and train maintenance personnel. These processors are becoming simple to design, construct, and maintain due to the advances in microprocessors and LSI chips such as the TEC1003J [TRW, 1977], which is a combination multiplier/accumulator.

4.5.2 Solving $MW = \overline{S}$

In order to solve $MW = \overline{S}$ in 15 milliseconds (msec) requires a 600-megaflops machine. This goal is unrealistic on a sequential machine in the near future. One reason for the high calculation rate is that a new steering vector is used during sweep mode every 15 msec, thereby requiring the calculation of new weights every 15 msec. As pointed out in the PEPE section, with O(N) parallel processors, the time to solve the weights for one steering vector is the same as the time to solve the weights for K steering vectors. By using this fact, we are no longer limited by the 15-msec time but instead by the estimate of how long M reflects the environment. For example, assume

$N = 200$,
$T_S$ (time per steering vector) = 15 msec,
$T_M$ (time M is good estimate of environment) = 1 sec, and
$T_W$ = time to solve for weights and K = number of weights to solve.

Then by solving

$$T_W = KT_S = T_M$$

for $T_W$ and K, we will determine the speed needed for the solution of $MW = \overline{S}$ and the number of steering vectors to solve for. This implies that a parallel processor which is slower than a sequential or vector processor may alternately be able to solve the total problem faster.

Because of the above considerations, we feel that a parallel approach is the most desirable and the most likely to be able to perform the task.

The next questions concern the number of processors and interconnect structure. As mentioned in Section 3.2.7, a technique with $10^{11}$ processors exists for N = 200; we believe this technique is ludicrous.

The techniques described in Appendix K with $O(N^2)$ processors seem reasonable and fast for small N. For N = 200, $N^2$ = 40,000, which is large for current technology. During the last few years, due to the microprocessor, machines have been proposed with hundreds and even 1,000 processors (see bibliography). The proposers of these machines have not built them and cite reliability problems to limit the number of processors under 1,000 until the 1980's. We believe that many of these limitations are due to the general-purpose nature of these proposed systems; however, many problems are simply a function of scale and will not be overcome with special-purpose processors.

We feel that machines of O(N) processors are viable today and that such machines can be made to run faster than sequential or vector processors. These parallel machines would also be cheaper to produce and maintain than sequential or vector processors of corresponding speed. As demonstrated in Section 4.4.2, the Gauss-Jordan algorithm is the best algorithm for a machine with O(N) processors, with each processor having sufficient memory and a global register or processor interconnect structure to simulate a global-register capability.

Another viable size is p, such that $p < N$. Implementations for both of these machine sizes were discussed in the PEPE section. We feel that the PEPE is the best machine among those examined for this problem. The PEPE program is very easy to read and follow, which is desirable from a software engineering aspect. There is also no wasted movement of data. One reason that the PEPE's program is clean is that each processor has sufficient memory. Examination of the STARAN's program for GJ shows that a problem lies in insufficient memory (see Section 4.4.4).

If a new machine is not designed, a parallel machine with asynchronous processors can be constructed out of minicomputers or array processors, such as the AP120B mentioned earlier. In this case, each processor would work independently on m rows of M, and then synchronize to receive or transmit a new global value. This is similar to the ILLIAC IV methods, except that the ILLIAC IV processors do not require synchronization.

# BIBLIOGRAPHY


Aho, A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, New York, 1974.

AP-120B Array Transform Processor: Processor Handbook, Floating Point Systems, Inc., 7259-02, Portland, Oregon, May 1976.

"Appendix J: Solution of Linear Systems," in Applications Study, Goodyear Aerospace Corporation, GER-16227, Akron, Ohio.

Bernhard, W. H., A Matrix Inversion Program for ILLIAC IV, Department of Computer Science, ILLIAC IV Document No. 184, University of Illinois at Urbana-Champaign, July 8, 1969.

Berra, P. B., and A. K. Singhania, Timing Figures for Inverting Large Matrices Containing Complex Numbers Using the Staran Associative Processor, Rome Air Development Center, RADC-TR-76-339, Griffiss Air Force Base, New York, November 1976, A034266.

Berra, P. B., and A. Singhania, Timing Figures for Inverting Large Matrices Using the Staran Associative Processor, Rome Air Development Center, RADC-TR-75-73, Griffiss Air Force Base, New York, March 1975, A009643.

Borodin, A., and I. Munro, The Computational Complexity of Algebraic and Numeric Problems, American Elsevier Publishing Company, Inc., New York, 1975.

Brennan, L. E., and I.S. Reed, "Theory of Adaptive Radar," IEEE Trans. on Aerospace and Electronic Systems, Vol. AES-9, No. 2, 1973, pp. 237-252.

Bunch, J.R., and J. E. Hopcroft, "Triangular Factorization and Inversion by Fast Matrix Multiplication," Mathematics of Computation, Vol. 28, No. 125, January 1974.

Calahan, D. A., W. N. Joy, and D. A. Orbits, Preliminary Report on Results of Matrix Benchmarks on Vector Processors, Systems Engineering Laboratory, SEL Report No. 94, University of Michigan, Ann Arbor, May 24, 1976.

Carr, R., Gauss-Seidell on ILLIAC IV, Department of Computer Science, ILLIAC IV Document No. 67, University of Illinois at Urbana-Champaign, May 1, 1967.

Chazan, D., and W. Miranker, "Chaotic Relaxation," in Linear Algebra and Its Applications, Vol. 2, 1969, pp. 199-222.

Cm* Review, edited by S. H. Fuller, A. K. Jones, and I. Durham, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1977.

A Collection of Papers on Cm*: A Multi-Microprocessor Computer System, edited by S. H. Fuller, et al., Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, February 1977.

"Comparison of the Basic STARAN Architecture with Architecture of Enhanced STARAN," Goodyear Aerospace Corporation, unpublished notes.

Comtre Corporation, Multiprocessors & Parallel Processing, edited by P. H. Enslow, Jr., John Wiley & Sons, New York, 1974.

Conrad, V., and Y. Wallach, "Iterative Solution of Linear Equations on a Parallel Processor System," IEEE Trans. on Computers, Vol. C-26, No. 9, 1977, pp. 838-847.

Control Data 7600 Series CYBER 70/Model 76 Computer Systems: Hardware Manual, Control Data Corporation, Arden Hills, Minnesota, 1975.

Control Data STAR-100 Computer: Hardware Reference Manual, Control Data Corporation, St. Paul, Minnesota, 1975.

CRAY-1 Computer System: An Introduction to Vector Processing, Cray Research, Inc., Bloomington, Minnesota, 1976.

CRAY-1 Computer System: Reference Manual, Cray Research, Inc., Publication 2240004, Bloomington, Minnesota, 1976.

Csanky, L., "Fast Parallel Matrix Inversion Algorithms," SIAM J. Comput., Vol. 5, No. 4, 1976, pp. 618-623.

Csanky, L., "On the Parallel Complexity of Some Computational Problems," Ph.D. dissertation, Computer Science Division, University of California, Berkeley, 1974.

Donnelly, J. D. P., "Periodic Chaotic Relaxation," in Linear Algebra and Its Applications, Vol. 4, 1971, pp. 117-128.

Edgar, D. S., Matrix Inversion and Iterative Refinement, Department of Computer Science, ILLIAC IV Document No. 141, University of Illinois at Urbana-Champaign, June 27, 1968.

Falk, Howard, "Illiac Then and Now", IEEE Spectrum, Vol. 13, No. 10, October 1976, pp. 64-69.

Flynn, M., "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Computers, Vol. C-21, No. 9, 1972, pp. 948-960.

Foster, C. C., Content Addressable Parallel Processors, Van Nostrand Reinhold Company, New York, 1976.

Gabriel, W. F., "Adaptive Arrays--An Introduction," Proc. of the IEEE, Vol. 64, 1976, pp. 239-272.

Gilmore, P. A., "Matrix Computations on an Associative Processor," in Lecture Notes in Computer Science, Vol. 24: Parallel Processing, edited by G. Goos and J. Hartmanis, Springer-Verlag, New York, 1975.

Githens, J. A., "An Associative, Highly-Parallel Computer for Radar Data Processing," in Parallel Processor Systems, Technologies, and Applications, edited by L.C. Hobbs, et al., Spartan Books, New York, 1970, pp. 131-149.

Han, L., Storage Schemes for Symmetrical Matrices, ILLIAC IV Document No. 95, University of Illinois at Urbana-Champaign, August 8, 1967.

Hopcroft, J. E., and L. R. Kerr, "On Minimizing the Number of Multiplications Necessary for Matrix Multiplication," SIAM J. Applied Math., Vol. 20, No. 1, 1971, pp. 30-36.

IEEE Trans. on Computers, Vol. C-26, No. 2, 1977 (Special Issue on Parallel Processors and Processing).

ILLIAC IV: Systems Characteristics and Programming Manual, Burroughs Corporation, Defense, Space and Special Systems Group, May 16, 1972.

Introduction to Burroughs Scientific Processor, Burroughs Corporation, July 1977.

Isaacson, E., and H. B. Keller, Analysis of Numerical Method, John Wiley & Sons, New York, 1966.

Johnson, P. M., CRAY-1 Computer System: An Introduction to Vector Processing, Cray Research, Inc., Publication 2240002, Minneapolis, Minnesota 1976.

Katz, J. H., "Matrix Computations on an Associative Processor," in Parallel Processor Systems, Technologies, and Applications, edited by L. C. Hobbs, et al., Spartan Books, New York, 1970., pp. 131-149.

Keller, R. M., "Look-Ahead Processors," ACM Computing Surveys, Vol. 7, No. 4, 1975, pp. 177-195.

Klyuyev, V. V., and N. I. Kokovkin-Shcherbak, On the Minimization of the Number of Arithmetic Operations for the Solution of Linear Algebraic Systems of Equations, translated by G. I. Tee, Computer Science Department, Technical Report C524, Stanford University, June 14, 1965.

Knowles, M., B. Okawa, Y.Muroaka, and R. Wilhelmson, Matrix Operations on ILLIAC IV, Department of Computer Science, ILLIAC IV Document No. 52, University of Illinois at Urbana-Champaign, March 1, 1967.

Lawrie, D. H., "Access and Alignment of Data in an Array Processor," IEEE Trans. on Computers, Vol. C-24, No. 12, 1975, pp. 1145-1154.

McDaniel, L. M., A Jacobi Algorithm for ILLIAC IV, Center for Advanced Computation, Document No. 21, University of Illinois at Urbana-Champaign, November 8, 1971.

McDaniel, L. M., Symmetric Decomposition of Positive Definite Band Matrices and the Corresponding Solution of Systems of Linear Equations on ILLIAC IV, Center for Advanced Computation, Document No. 34, University of Illinois at Urbana-Champaign, July 1, 1972.

Miranker, W. L., "A Survey of Parallelism in Numerical Analysis," SIAM Review, Vol. 13, No. 4, 1971, pp. 524-547.

Ogura, M., A Parallel Scheme for Reducing a Real Matrix to the Upper Hessenberg Form, Center for Advanced Computation, Document No. 11, University of Illinois at Urbana-Champaign, September 1, 1971.

Pace, S. A., An ILLIAC IV Gaussian Elimination Package for Non-Core-Contained Matrices, Center for Advanced Computation, Document No. 40, University of Illinois at Urbana-Champaign, September 1, 1972.

Pease, M. C., The Indirect Binary n-Cube Microprocessor Array, Stanford Research Institute, Menlo Park, California, 1975.

Pease, M. C., "Matrix Inversion Using Parallel Processing," J. of the Association for Computing Machinery, Vol. 14, No. 4, 1967, pp. 757-764.

Proceedings of the First Annual Symposium on Computer Architecture, December 9-11, 1973; Conference Proceedings of the 2nd Annual Symposium on Computer Architecture, January 20-22, 1975; Conference Proceedings for the 3rd Annual Symposium on Computer Architecture, January 19-21, 1976; Conference Proceedings of the 4th Annual Symposium on Computer Architecture, IEEE Computer Society Publications Office, Long Beach, California.

Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing; Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing; Proceedings of the 1976 International Conference on Parallel Processing; IEEE Computer Society, Long Beach, California.

A Prospectus for High Speed Computing, Phoenix Project, Memorandum No. 006, Institute for Advanced Computation, Sunnyvale, California, February 25, 1977.

Ramamoorthy, C. V., and H. F. Li, "Pipeline Architecture," ACM Computing Surveys, Vol. 9, No. 1, 1977, pp. 61-102.

Reddaway, S. F., "DAP--A Distributed Array Processor," in Conference Proceedings of the First Annual Symposium on Computer Architecture, December 9-11, 1973, pp. 61-65.

Reed, I. S., J. D. Mallett, and L. E. Brennan, "Rapid Convergence Rate in Adaptive Arrays," IEEE Trans. on Aerospace and Electronic Systems, Vol. AES-10, No. 6, 1974, pp. 853-863.

Rosenfeld, J. L., "A Case Study in Programming for Parallel-Processors," Communications of the ACM, Vol. 12, No. 12, 1969, pp. 645-655.

Sameh, A. H., and R. H. Bezdek, Methods for Increasing the Computational Efficiency of Input-Output and Related Large Scale Matrix Operations, Center for Advanced Computation, Document No. 66, University of Illinois at Urbana-Champaign, revised May 1973.

Sameh, A. H., and D. J. Kuck, Linear System Solvers for Parallel Computers, Department of Computer Science, Report No. UIUCDCS-R-75-701, University of Illinois at Urbana-Champaign, February 1975.

Sameh, A. H., and D. J. Kuck, "A Parallel QR Algorithm for Symmetric Tria-diagonal Matrices," IEEE Trans. on Computers, Vol. C-26, No. 2, 1977, pp. 147-153.

Sastry, K. V., and R. Y. Kain, "On the Performance of Certain Multiprocessor Computer Organizations," IEEE Trans. on Computers, Vol. C-24, No. 11, 1975, pp. 1066-1074.

Schaffer, K. L., "Associative-Parallel Applications to Radar Signal Processing," in Proceedings of the 1976 International Conference on Parallel Processing, August 24-27, 1976, pp. 145-153.

Shapard, J. M., D. J. Edelblute, and G. L. Kinnison, Adaptive Matrix Inversion, Naval Undersea Research and Development Center, NUC TN 528, San Diego, California, May 1971.

Shedler, G. S., "Parallel Numerical Methods for the Solution of Equations," Communications of the ACM, Vol. 10, No. 5, 1967, pp. 286-291.

STARAN Reference Manual, Goodyear Aerospace Corporation, GER-15636B, Akron, Ohio, September 1974.

Stevens, J. E., Jr., Matrix Multiplication Algorithms for ILLIAC IV, Department of Computer Science, ILLIAC IV Document No. 231, University of Illinois at Urbana-Champaign, August 26, 1970.

Strassen, V., "Gaussian Elimination is Not Optimal," Numerische Mathematik, Vol. 13, 1969, pp. 354-356.

Thurber, K. J., Large Scale Computer Architecture: Parallel and Associative Processors, Hayden Book Company, Inc., Rochelle Park, New Jersey, 1976.

Thurber, K. J., and L. D. Ward, "Associative and Parallel Processors," ACM Computing Surveys, Vol. 7, No. 4, 1975, pp. 215-255.

Troy, J. L., Real-Time Advanced Data Processing Parallel Element Processing Ensemble (PEPE): PEPE Hardware Reference Manual, System Development Corporation, TM-HU-051/001/00, 1 February 1977.

Troyer, S. R., A TRANQUIL Program to Invert a Matrix by Partitioning, Department of Computer Science, ILLIAC IV Document No. 179, University of Illinois at Urbana-Champaign, April 1, 1968.

TRW LSI Multiplier-Accumulator, TRW LSI Products, Redondo Beach, California, August 1977.

Varga, R. S., Matrix Iterative Analysis, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1962.

Westlake, J. R., A Handbook of Numerical Matrix Inversion and Solution of Linear Equations, John Wiley & Sons, Inc., New York, 1968.

Wilkinson, J. H., Rounding Errors in Algebraic Processes, Prentice-Hall, Englewood Cliffs, N.J., 1963.

Wilkinson, J. H., and C. Reinsch, Handbook for Automatic Computation. Vol. II: Linear Algebra, Springer-Verlag, New York, 1971.

Wilks, S. S., "Problem 12. 24," in Mathematical Statistics, John Wiley & Sons, New York, 1962, p. 392.

Young, D. M., "On the Solution of Large Systems of Linear Algebraic Equations with Sparse, Positive Definite Matrices," in Numerical Solution of Systems of Nonlinear Algebraic Equations, edited by George D. Byrne and Charles A. Hall, Academic Press, New York, 1973.

# *MISSION*
## *of*
## *Rome Air Development Center*

*RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications ($C^3$) activities, and in the $C^3$ areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.*