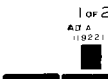
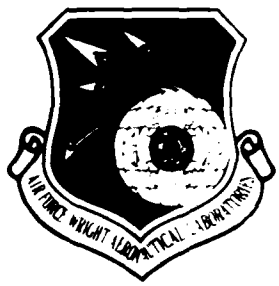
AD-A119 221 AIR FORCE WRIGHT AERONAUTICAL LABS WRIGHT-PATTERSON AFB OH F/G 12/1
SOFTWARE OPTIMIZATION FOR ARRAY PROCESSORS. AN AP-120B KALMAN F--ETC(U)
JUN 82 E C DUDZINSKI
UNCLASSIFIED AFWAL-TR-82-1100 NL



AFWAL-TR-82-1100

SOFTWARE OPTIMIZATION FOR ARRAY PROCESSORS
AN AP-120B KALMAN FILTER



Edward C. Dudzinski
Reference Systems Branch
System Avionics Division

June 1982

Final Report for Period 1 April 1981 - 28 May 1982

Approved for public release; distribution unlimited.

AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

EDWARD C. DUDZINSKI
Project Engineer

RONALD L. RINGO
Chief, Reference Systems Branch
System Avionics Division

FOR THE COMMANDER

RICHARD H. BOIVIN, Colonel, USAF
Chief, System Avionics Division
Avionics Laboratory

"If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAN, *W-PAFB, OH 45433 to help us maintain a current mailing list".*

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1 REPORT NUMBER: AFWAL-TR-82-1100	2 GOVT ACCESSION NO: AD-A117 221
3 RECIPIENT'S CATALOG NUMBER:	
4 TITLE (and Subtitle): Software Optimization for Array Processors An AP-120B Kalman Filter	5 TYPE OF REPORT & PERIOD COVERED: Final Report for Period 1 April 81 to 28 May 82
	6 PERFORMING ORG. REPORT NUMBER:
7 AUTHOR(s): Edward C. Dudzinski	8 CONTRACT OR GRANT NUMBER(s):
9 PERFORMING ORGANIZATION NAME AND ADDRESS: Avionics Laboratory (AFWAL/AAAN) AF Wright Aeronautical Laboratories, AFSC Wright-Patterson AFB, OH 45433	10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 60951533 E2204F
11 CONTROLLING OFFICE NAME AND ADDRESS:	12 REPORT DATE: 1 June 1982
	13 NUMBER OF PAGES: 120
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):	15 SECURITY CLASS. (of this report): UNCLASSIFIED
	15a DECLASSIFICATION/DOWNGRADING SCHEDULE:

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)

18 SUPPLEMENTARY NOTES

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

Array Processors	Code Optimization
Kalman Filters	Pipelining
Parallel Processing	Execution Timing
AP-120B	

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

Aircraft navigation imposes critical speed requirements on the embedded avionics processor, requirements which will become even more rigorous in the future as additional, increasingly sophisticated navigation data becomes available in the cockpit. The fusion of navigation sensor data to arrive at an accurate position determination is done using a programmed algorithm called a Kalman filter.

Achieving the necessary processing speed for next-generation navigation filters will require the use of innovative machine architectures. One of

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

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

the most promising configurations, given the computational nature of the Kalman filter, is the array processor. To explore the software issues and demonstrate the potential speedup made possible by an array architecture, an in-house research project was undertaken to install a Kalman filter on a vector machine and maximize its execution speed. The actual hardware consisted of a DEC PDP 11/70 host for initialization with a slave FPS AP-120B array processor to execute the filter algorithm. In order to assess software optimization techniques, processing times for a simulated scenario were measured initially with the AP-120B programmed to function as if it were a strictly serial processor, and then again after the software had been optimized to exploit the machine's parallel architecture.

Algorithm restructuring, expression-tree height reduction, maximization of loop parallelism, and other techniques were applied to the Kalman filter algorithm chosen for this project. Although software development was a formidable task, execution speed was increased by a factor of five for the five-state sample filter, with an extrapolated speedup of greater than eight times for a more characteristic twenty-element filter. Array architectures are a powerful resource for meeting avionics processing needs, given the requisite software optimization expertise.

FOREWORD

The software development documented in this report was performed as an in-house project for the Reference Systems Analysis and Evaluation Group, Systems Avionics Division, Air Force Avionics Laboratory, Wright-Patterson Air Force Base. The work was done between April 1, 1981 and May 28, 1982 under work unit 60951533, Evaluation of the Use of a Parallel Processor for Navigation Filtering.

Implementation of a Kalman filter on an array processor was first suggested to me by my co-worker, Mr. Pete Howe. My supervisor, Mr. William Shephard, then provided exceptional support in allowing me to define the project and carry out the research.

The Kalman filter implemented for this optimization study was designed by Mr. Stan Musick, who also works in the Reference Systems Branch. Stan provided guidance on Kalman theory and wrote the FORTRAN version of the filter algorithm. That FORTRAN code is used as a test problem for a tool that Mr. Musick designed called Simulation for Optimal Filter Evaluation (SOFE).

In the Kalman filter explanation of Section 3, much

reliance is placed on an example problem that was taken from Dr. Pete Maybeck's text, Stochastic Models, Estimation, and Control, Volume 1 (Academic Press, 1979).

Installing the filter on the PDP 11/70 host with its attached AP-120B required not only support, but heroic efforts from system manager Jim Barnes. A multitude of hardware problems were quickly and competently handled. And Dr. Bob Phelps of the Laboratory's Radar Division troubleshot errors in the AP-120B system software, literally making it possible to use the array processor.

Miss Denise Klawonn typed the lengthy text and often worked late so that deadlines could be met.

TABLE OF CONTENTS

Table of Contents (Cont'd)

LIST OF ILLUSTRATIONS

List of Illustrations (Cont'd)

LIST OF TABLES

SECTION I

INTRODUCTION

Execution speed is a primary consideration in many software applications. Signal processing, interactive graphics, and speech recognition all require real-time response under a considerable computational load. Aircraft navigation is another application with critical speed requirements, requirements which will become even more rigorous in the future as additional, increasingly sophisticated navigation data becomes available in the cockpit. The fusion of navigation sensor data to arrive at an accurate position determination is done using a programmed algorithm called a Kalman filter.

Achieving the necessary processing speed for next-generation navigation filters will require the use of innovative machine architectures. One of the most promising configurations, given the computational nature of the Kalman filter, is the array processor. To explore the software issues and demonstrate the potential speedup made possible by an array architecture, a project was undertaken to install a Kalman filter on a vector machine and maximize its execution speed. The actual hardware consisted

of a PDP 11/70 host for initialization with a slave AP-120B array processor to execute the filter algorithm. In order to assess software optimization techniques, processing times for a simulated scenario were measured initially with the AP-120B programmed to function as if it were a strictly serial processor, and then again after the software had been optimized to exploit the machine's parallel architecture

The Kalman filter algorithm itself will be examined in depth in Section 3 of this report, but the structure of its operations can be simply described. A vector is constructed of elements that estimate the quantities of interest, such as an aircraft's altitude or velocity. Those estimates are initialized to known values, and the filter attempts to keep them current during takeoff, flight and landing. It does this by means of two separate operations. First, instrument readings are incorporated into the estimates when they are available. Secondly, differential equations describing the aircraft's motion and the measurement devices are numerically integrated to keep the estimates current between measurement updates. Both the numerical integration and the measurement update consist of operations on vectors and square matrices, their dimensions being equal to the number of quantities estimated. Typical airborne navigation filters have to track about 20

"states", or elements, in order to calculate the aircraft's position.

In order to take maximum advantage of the independent, pipelined functional units of array processors such as the AP-120B, the algorithm of interest must be restructured to increase its vector nature. In essence, the architecture of the algorithm is matched to that of the machine. Assembly language code is then written using techniques such as expression-tree height reduction, overlap of independent computations, and the use of register cache memory to enhance the program's compactness and execution speed. Most critically, processing loops are optimized by identifying the minimum number of instructions dictated by hardware constraints, writing sequential code for loop computations, then "wrapping" that code into the minimum loop size.

All of these techniques, along with a few other tricks, were applied to the Kalman filter algorithm chosen for this project. Although software development was a formidable task, execution speed was increased by a factor of five for the five-state sample filter, with an extrapolated speedup of greater than eight times for a more characteristic twenty-element filter. Array architectures are a powerful resource for meeting avionics processing needs, given the requisite software optimization expertise.

SECTION II

ARRAY PROCESSORS

Vector processors can be most reliably defined as computing machines with an architecturally-endowed capability for more efficient operation on vectors than on scalars. In terms of the arithmetic logic unit, this is achieved in one of three ways.

1. Multiple ALU's operating simultaneously on different elements of a vector (Illiac IV),

2. Vector functional units that operate on an array of data (Cray), or

3. Pipelined processing elements that operate on scalars (AP-120B).

Pipelining works precisely like an assembly line. An operation such as a floating-point multiplication is broken down into discrete components, each requiring a single CPU cycle. Intermediate results are captured in buffers that separate the logic circuitry of these sub-operations, so that each clock cycle allows the introduction of new

operands into the functional unit. When a vector of elements can be "streamed" through such a pipelined unit, the second and successive results become available one clock cycle after the first, regardless of the overall functional unit time. The speedup of a k segment pipeline over a serial unit requiring k clocks is equal to:

$$S = T_S / T_P = (n*k)/(k + n - 1)$$

where n is the length of the vector. To illustrate, assume a multiplier requiring three CP cycles per operation (k=3) performs a scalar-vector multiplication on a vector of length ten (n=10). Clearly, this would require a total of

$$n*k = 10*3 = 30 \text{ CP cycles.}$$

On the other hand, a three-segment pipelined multiplier would require three cycles to generate the first result (k) while each successive answer would be available one cycle later. This gives a total operation time for a ten-element vector of

$$k + (n - 1) = 3+9 = 12$$

and a speedup equal to 30/12 or 2.5.

As the expression shows, the greater the length of a vector streamed through a pipeline, the greater the speedup. Figure 1 shows the floating multiplier pipeline of

$$S = T_S / T_P = N*K/(K + N - 1)$$

$$\underline{C} = M * \underline{X}$$

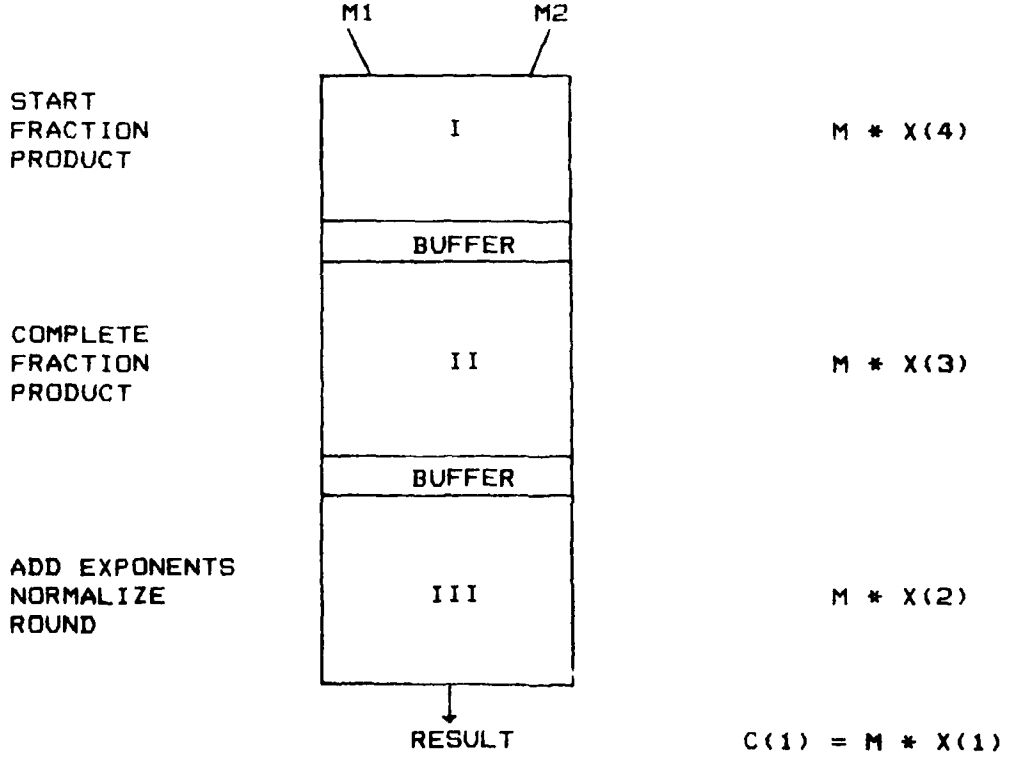


Figure 1. AP-120B Floating Multiplier Pipeline.

the AP-120B with segment functions to the left and a hypothetical computation at the right. This figure describes the situation three instruction cycles after initiating a vector-scalar multiplication (M*X). The first element result (C(1)) is available, having required three CP cycles to complete. The multiplication of the second element (M*X(2)) was begun one instruction after the first, so its partial result is currently in segment three, and will be available in the next CP cycle. Similarly, operations on elements three and four are already in process in the pipeline in segments two and one respectively. Calculation of C(1) required the complete functional unit time, while each successive result will be available one cycle after its predecessor.

The other primary requirement of an array processor is data flow circuitry to accomodate the increased computational throughput. Fast processing elements are useless if they sit idle during frequent memory transfers. In many machines, such as the Cray and AP-120B, this problem is solved with a large store of in-CPU cache registers. The AP also has multiple CPU buses which allow operand fetches from these registers to occur simultaneously with result storage into them. Intermediate results can be kept in this cache to eliminate a cycle of memory accesses. These cache registers can also be used to stage

data by fetching operands into them for future computations or writing previous results from them at the same time that the ALU is operating on other data.

1. FLOATING POINT SYSTEM'S AP-120B

The AP-120B manufactured by Floating Point Systems, Inc. of Portland, Oregon is a synchronous, parallel, pipelined array processor. It combines fast logic circuitry (167 nanosecond cycle time) with a highly parallel architecture and pipelined floating-point functional units to achieve tremendous speedup over conventional machines, especially in the case of highly vectorized problems. Figure 2 is a simple block diagram showing the functional units.

The multiplier is a three-segment pipeline that operates on the AP-120B's 38-bit floating-point number. The floating-point adder is a two-segment pipeline. Each segment requires a single clock cycle. Another ALU is available for integer operations on 16-bit representations, which are completed in a single CP cycle. All three of these functional units are entirely independent and can be utilized simultaneously. This means that the basic instruction issue speed of 6 MHrz can provide a theoretical

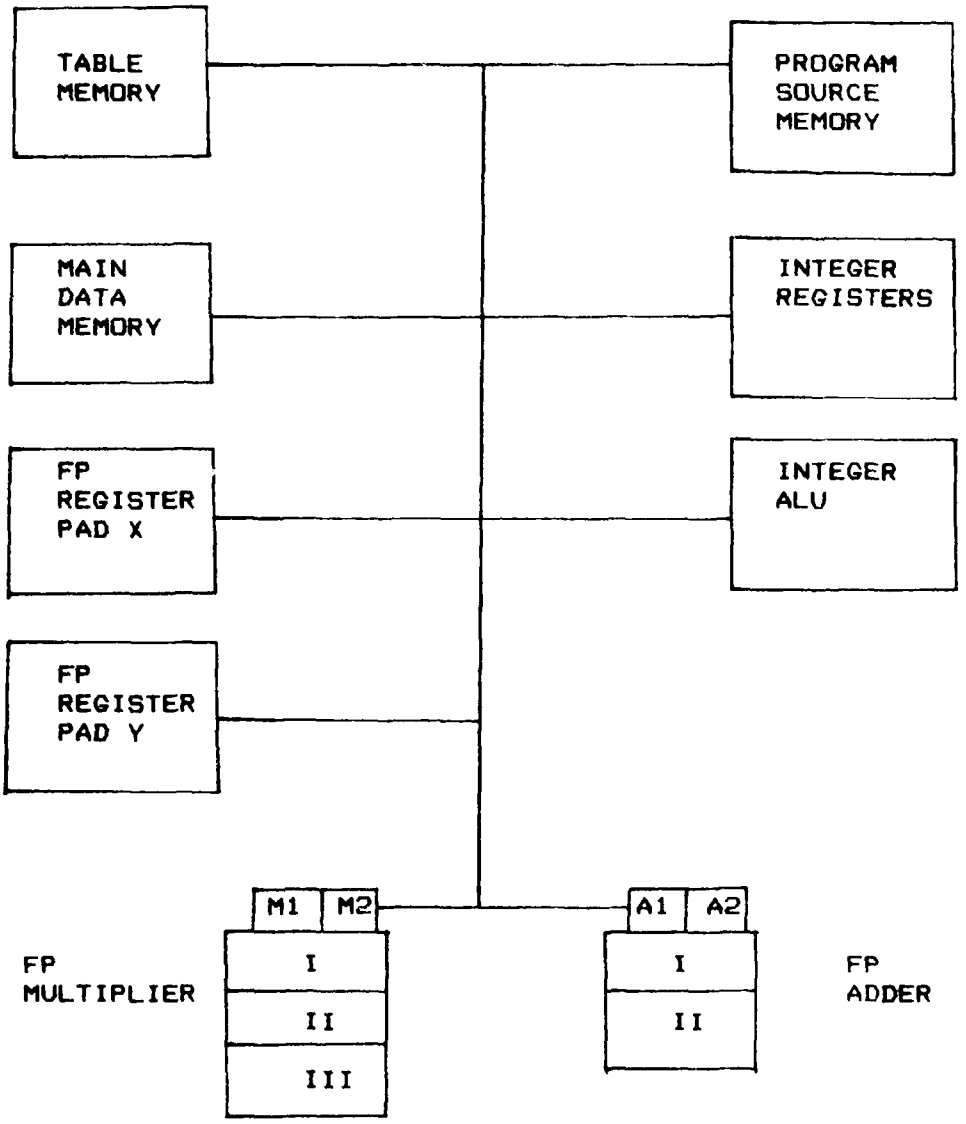


Figure 2. AP-120B Architecture.

computation rate of 12 MFLOPS, since each machine cycle can produce a result from both floating-point units. And since integer operations and overhead are done in parallel with the computations, that theoretical maximum can be more easily reached and sustained for a longer time.

There are 11 interconnected data buses in the CPU to accomodate simultaneous data flow to support the highly parallel architecture. An I/O bus carries data to and from physical devices external to the AP-120B. Four separate buses provide operands to the floating adder and multiplier, while two more buses make available the results of these units. Integer results are communicated via an S-Pad bus, and a general Data Pad bus services all but the floating-point units. Two more buses connect AP-120B functional units to registers that are used for communication with the host processor. Although these 11 buses are the key to AP usage by enabling data flow to support simultaneous operations, coordinating data flow is one of the difficult aspects of programming the machine.

Main data memory (MD) comes in fast and slow versions, and one, two or four banks for interleaved access. With either speed memory, cycle time is three clock cycles (500 ns) and a single bank access can be initiated every two cycles. The fast version of MD memory allows interleaved access every clock cycle, while slow MD restricts

access to every other cycle. Data is transferred between host and AP via dedicated DMA lines, with reformatting accomplished on the fly.

A table memory unit (TM) in the CPU stores commonly used constants that can be fetched with a cycle time of only 2 clocks (333 ns). ROM TM is standard, and RAM is available.

Instructions are stored entirely in a separate Program Source memory (PS) eliminating the need for MD access unless overlays are used. Loading of the next instruction into the control buffer is overlapped with decoding of the current instruction in order to achieve 6 MIPS. Figure 3 shows a block diagram of the control unit. The "+1" in the figure simply indicates that the Program Source Address register is automatically incremented unless an instruction is decoded that explicitly alters it.

The AP-120B configuration used for this project had 2.5K of ROM table memory, 1K of RAM table memory, and 1K of program storage memory. MD memory was the fast version, with 32K available.

The instruction word itself is sixty-four bits long, with six separate op-code groups allowing ten functions to be specified and executed simultaneously. In a single instruction word it is possible to initiate a floating-point multiplication and addition, an integer operation, an MD

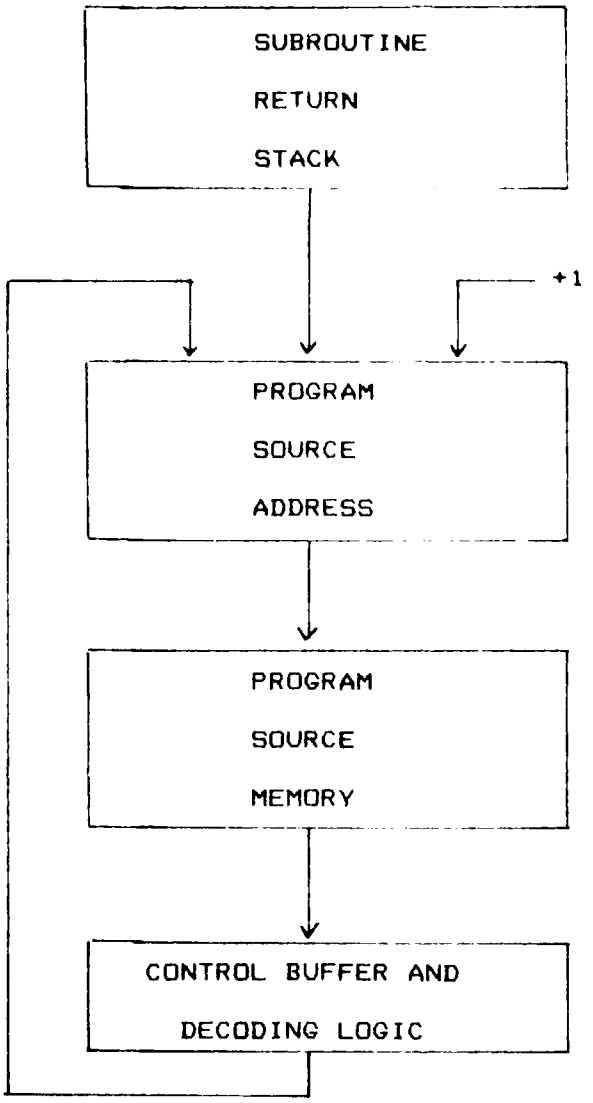


Figure 3. AP-120B Control Unit.

memory access, a table memory access, stores to both data pads and fetches from both, and a branch operation. As mentioned above, the independent busing circuitry supports these simultaneous operations, with some limiting constraints. The op-code field breakdown appears in Figure 4. Overlapped field descriptions indicate operations that use the same instruction bits. Hence, if you use an op-code requiring an immediate value in bits 48 through 63, you cannot specify operations from the multiply or memory groups in that same instruction word.

Floating-point format in the AP-120B consists of a ten bit biased integer exponent and a twos-complement twenty-eight bit mantissa. This word-size, coupled with a convergent rounding algorithm in the floating adder and multiplier, provides a precision of 8.1 decimal digits. The normalized binary representation supports a floating-point range from $3.7*10^{-155}$ to $6.7*10^{+153}$.

Communication between AP and host (in this case a PDP 11/70) is accomplished via a virtual panel of host accessible registers. Bits in these registers are explicitly set or cleared to initiate or terminate DMA data transfer, object module loads, program execution, and all other AP-120B processor functions. Certain of the registers can be read to determine when a DMA operation or a routine execution has terminated. Each register in this virtual

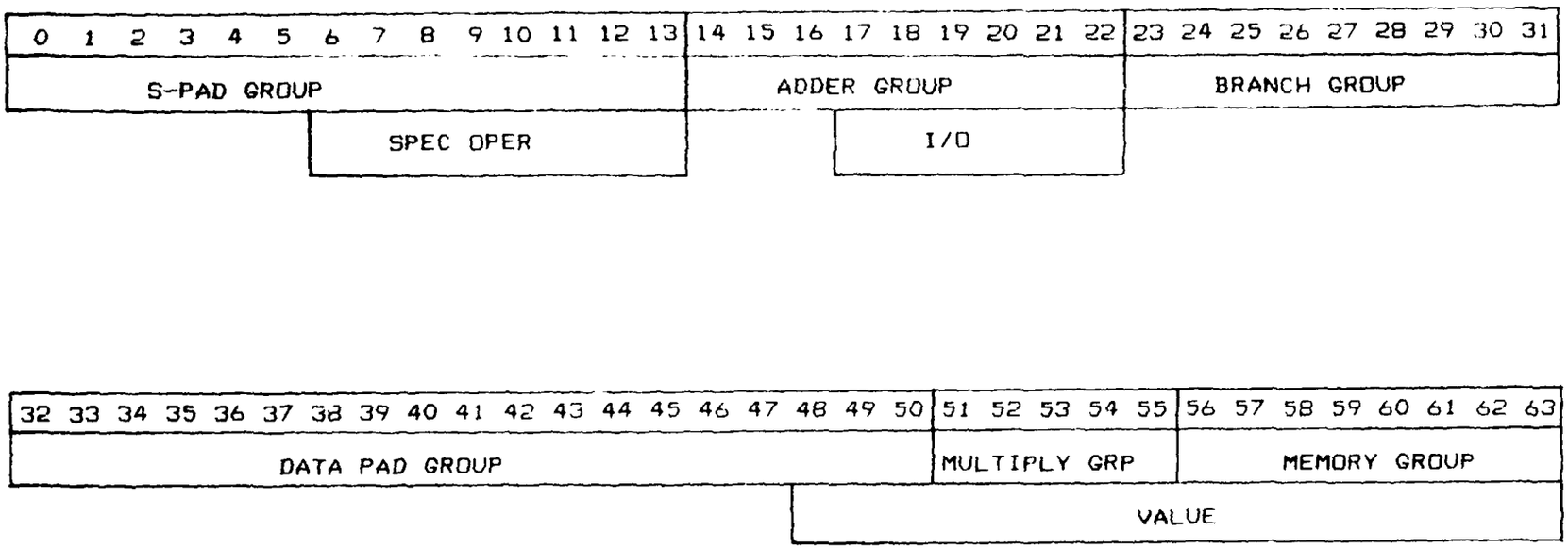


Figure 4. AP-120B Instruction Word

panel has an address on the host bus, enabling direct assembly language access. When the host program is written in FORTRAN, the registers are accessed and the AP-120B controlled via a library of FORTRAN-callable executive subroutines.

2. AP-120B SUPPORT SOFTWARE

Although the documentation leaves a little to be desired, a full complement of host-resident support software is provided with the AP. To illustrate, the program development stages will be examined. Figure 5 provides a map of these stages.

First, a host FORTRAN program is written to initialize data, perform any executive computations, control AP operation, and transfer data between the two processors. This data transfer can be done via FORTRAN-like parameter-passing or explicit calls to AP executive routines. Specifically, calls to APPUT transfer data from the 11/70 main memory to an indicated AP MD address, and calls to APGET fetch data back once execution is complete. A call to the AP-resident object module initiates AP execution.

Routines that will execute on the AP-120B are written

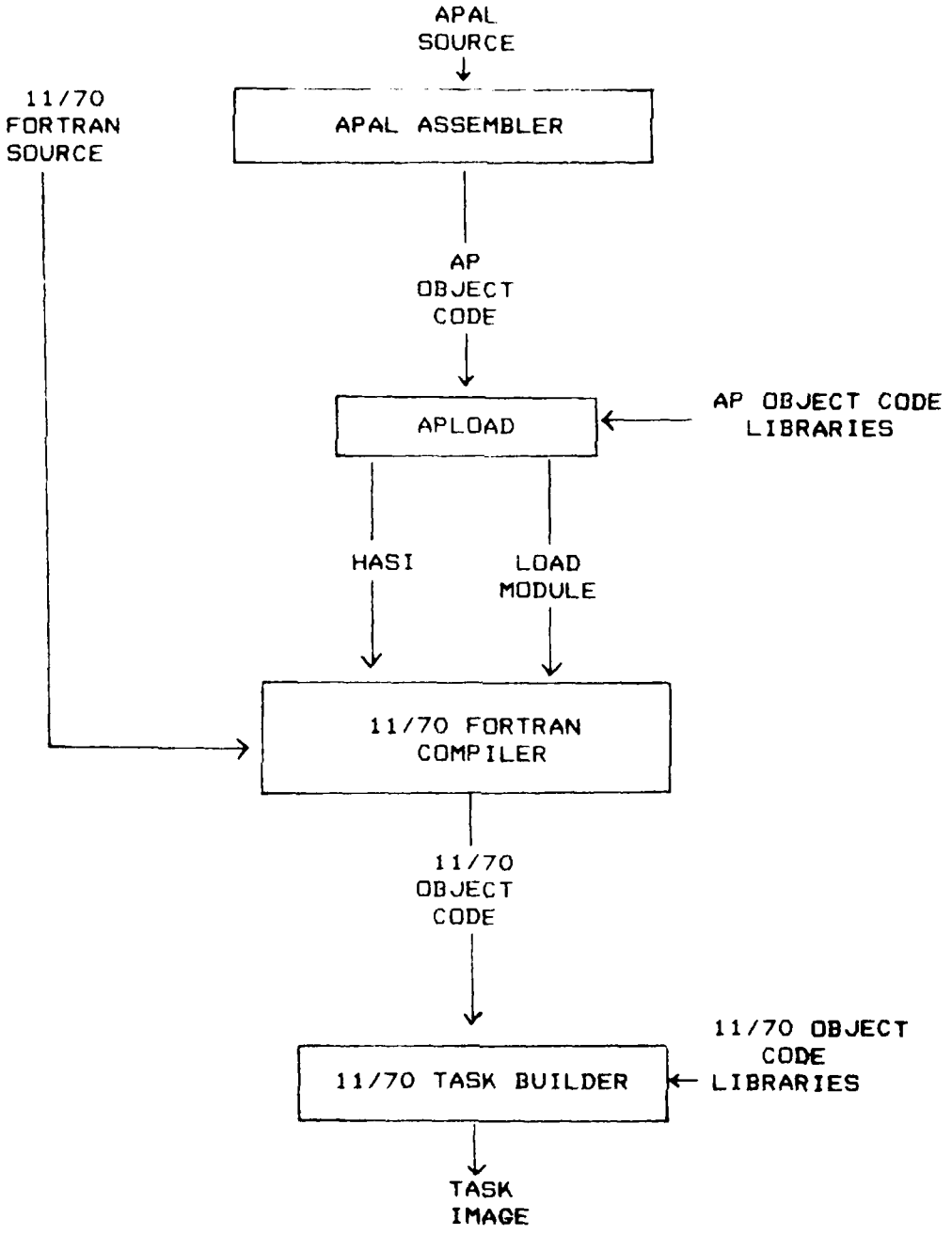


Figure 5. AP-120B Program Development

in AP Assembly Language, run through a cross-assembler to generate object modules, then linked with an AP system program called APLOAD. This linker generates two output files, both of which are normally FORTRAN subroutines The first output file is termed the HASI file, standing for Host AP Software Interface. It is a FORTRAN subroutine with the same name as the AP-resident program that is called from the host FORTRAN. In fact, when calling the AP routine, what is actually called is the HASI. The HASI code checks to see whether or not the necessay object module is loaded into AP Program Source memory. If it is not, the HASI calls the other FORTRAN subroutine created by the linker, and this second routine loads the object module into the AP. The object module itself is stored as data statements in that second subroutine. Once the AP module is loaded, the HASI transfers control to it.

If the AP object module is too large to be accomodated in the host main memory, it can be converted into a disk-resident binary file that is not a FORTRAN subroutine. To load this file, an AP executive call is made from the host FORTRAN assigning the load module a logical unit number, and it is then directly loaded by the HASI subroutine.

Figure 6 shows a FORTRAN listing of the HASI controlling execution of the Kalman filter. Routine APLMLD loads

a disk-resident object module into the AP. It also sets IDLM equal to 1 so if this AP routine is again called from the 11/70 FORTRAN, it will not be reloaded. APRUN initiates AP execution. APEXC is a dummy call for future use by the manufacturer, and APOVLD will be explained later.

```
      SUBROUTINE APCTL
       COMMON /APLDCM/ IPAV( 33),NU2,IDLM,NU1,
     *  IPPAAD,IPPAND,IOVS(33),LMT(10,3),LMTE
      COMMON /CODE 1/ CODE
      INTEGER CODE(   200)
      IF(IDLM.NE.      1)CALL APLMLD(      1,CODE,    200)
      IPAV(1)=  0
      CALL APOVLD (      1)
      CALL APRUN(   100,      0,      2,      1,13)
      CALL APEXC
      RETURN
      END
       BLOCK DATA
       COMMON /APLDCM/ IPAV( 33),NU2,IDLM,NU1,
     *  IPPAAD,IPPAND,IOVS(33),LMT(10,3),LMTE
       DATA NU2,IDLM,NU1,IPPAAD,IOVS(2),LMTE
     *  /0,0,0,0,0,0/     ,
       END
```

Figure 6. HASI Code for an AP Disk-Resident Object Module

Finally, all the object modules are linked with the 11/70 Task Builder to create a task image file. The object modules include the 11/70 FORTRAN routines, the AP HASI, and the AP load module if it is not disk resident. AP executive routine references are satisfied from a library of 11/70 object code. The resulting task is run normally, with control shifting between the 11/70 and the AP-120B as AP subroutines are called and executed.

Two more system programs are provided for off-line and hardware debugging. A host-resident simulator allows exercise of the AP routines with the usual repertoire of simulator controls such as breakpoints and single-step execution. This simulator exactly reproduces the AP timing and numerical results. Another software interface allows execution on the hardware itself with many of the same debugging aids that the simulator provides. Both of these aids are invaluable when it comes to debugging APAL code.

One last support software issue was raised by the limited program source memory on our AP. Although the linear Kalman code fit into the 1K of PS memory as a single object module, the optimized version was too large by almost 200 words. Two options were available to resolve the problem.

First, the code could be broken into two or more FORTRAN-callable object modules with the filter executive run in the 11/70, or internal AP overlays could be used. Since the overlay documentation was virtually non-existent, the former approach was implemented first.

Although the use of two separate load modules called from the 11/70 FORTRAN program worked well enough, the overhead was prohibitive as might be expected. In one test scenario, better than 93% of the total AP execution time was spent swapping load modules between 11/70 disk

and AP-120B PS, and transferring execution control between host and AP. This seemed unacceptable, so overlays were attempted.

After trial-and-error discovery of undocumented register usage by the overlay routine, lack of cross-referencing between overlays for utility routines, linker idiosyncracies in overlay segment MD placement, and a variety of other software bugs, overlay generation and use was successfully utilized. In the HASI code of Figure 6, the APOVLD call loads the root overlay segment from MD into PS memory.

Figure 7 shows the APLOAD input that was necessary to define and create an overlay tree structure. Note the three different library references necessary because of the overlay segments' independence. Documented linker commands to force utility subroutines into the always-resident root segment did not work, library subroutine externals could only be satisfied from within the overlay segment executing.

Table 1 shows timing results for execution using three different object module configurations: a single AP modu'e, two separate modules swapped from the 11/70 disk memory under FORTRAN control, and a single object module using three overlay segments swapped between AP program source and main data memory. Obviously, overlay use was

effective.

```
OUTPUT PAPCTL.HAS PAPCTL.LM/D
TREE ((1 (2) (3)))
OVERLAY 1
CALL APCTL /
LOAD OAPCTL.APO
LOAD OVECTRN.APO
LOAD OOUT.APO
MDOFF 20000
MMAX 100000
LIB NLIB.OBJ
OVERLAY 2
LOAD OKUTMER.APO
LOAD ODERIV.APO
LOAD OFPPPFT.APO
LIB NLIB.OBJ
OVERLAY 3
LOAD OPSQRT.APO
LOAD OUPDATE.APO
LOAD OXSPLUS.APO
LOAD OSQRS.APO
LIB NLIB.OBJ
LINK
```

Figure 7. APLOAD Command Input for Overlay Generation

	EXECUTION TIME (seconds)		
TEST	SINGLE LOAD MODULE	TWO LOAD MODULES	INTERNAL AP OVERLAYS
1	6.18	7.24	7.22
2	2.10	3.12	3.28
3	7.13	48.50	9.04
4	3.03	44.27	5.08
5	0.23	0.32	1.48
6	62.28	96.77	61.95

Table 1. Timing Comparison of Object Module Configuration

SECTION III

KALMAN FILTERING

The topic of this paper is software techniques for array processors. Examples are drawn from and results are reported for the optimization of a Kalman filter algorithm. This section is intended to provide a brief introduction to the concepts and computations that make up a Kalman filter.

The observation and control of any complex physical system requires the processing of sensor measurements to estimate the important physical quantities of the system. These physical quantities are called the "states" of the system. Kalman filters combine sensor data and system models to estimate state values with minimum error.

A Kalman filter is an optimal, recursive data processing algorithm, typically implemented on a digital computer. By recursive we mean that previous measurements do not have to be saved for reprocessing, so memory requirements are known before operation and never change.

The accuracy of a Kalman filter depends on the quality of the preliminary analysis done to design it. Mathematical models, in the form of differential equa-

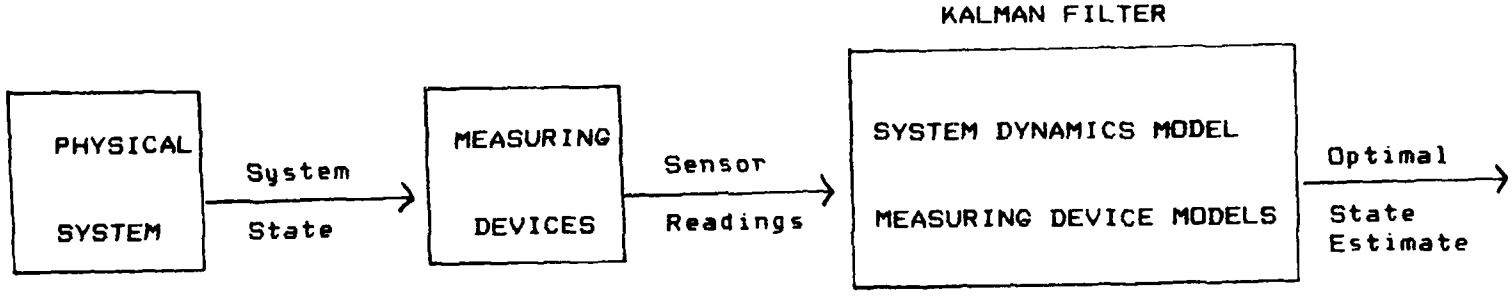


Figure 8. A Kalman Filter Estimator

tions, are built to simulate the dynamics of the system and the measurement devices. These models are simplified to meet the computational and memory size limitations of the hosting computer. Statistical descriptions of the system and measurement errors, of the system noises, and of the simplifying assumptions in the model are derived. Lastly, the filter is fine-tuned by analysis and simulation to find parameter values that minimize errors.

Once the system model is built, two stages make up the actual operation of the filter. Measurement updates incorporate sensor readings into the filter state estimates. Between measurements, the mathematical model is propogated over time so that the estimates are kept current. Time propogation in the filter of this project is done by a fifth-order Kutta Merson numerical integration of the descriptive differential equations.

Measurement update is best described by example. Variable names used in the example will follow these conventions: Scalars are lower case letters, never underlined. Vectors are lower or upper case letters, always underlined Matrices are upper case letters that are not underlined.

Our sample application, taken from Dr. Maybeck's text, is position determination in a single dimension. The system of interest is a boat, and the measurements are

star sightings.

We start, as we float offshore, by taking a sighting. This gives our measured position as z_1 at time t_1, and let's say we know our error, or variance, as $\delta^2_{z_1}$. Now our position estimate equals our measurement

(1) $\hat{x}(t_1) = z_1$,

(the "^" signifies an estimated quantity), and our position uncertainty is the measurement variance,

(2) $\delta^2_x(t_1) = \delta^2_{z_1}$.

At time $t_2 \tilde{=} t_1$, a more skilled observer takes a second sighting, getting position z_2 with a smaller error of $\delta^2_{z_2}$. We can update our position estimate with this new information. Figures 9 and 10 show plots of the conditional probability density of both measurements. These are Gaussian distributions giving the probability that any given position is the true one. The more accurate measurement has a higher, narrower peak showing a greater probability that the mean, equal to our actual measurement, is the true value. We can combine the information contained in both these measurements by constructing the conditional probability density of Figure 11, with a mean

(3) $\mu = [\delta^2_{z_2}/(\delta^2_{z_1} + \delta^2_{z_2})]\, z_1 + [\delta^2_{z_1}/(\delta^2_{z1} + \delta^2_{z_2})]\, z_2$

and a variance $\delta^2_x(t_2)$ where

(4) $1/\delta^2_x(t_2) = (1/\delta^2_{z_1}) + (1/\delta^2_{z_2})$.

This mean, μ, is the mean of the combined probability

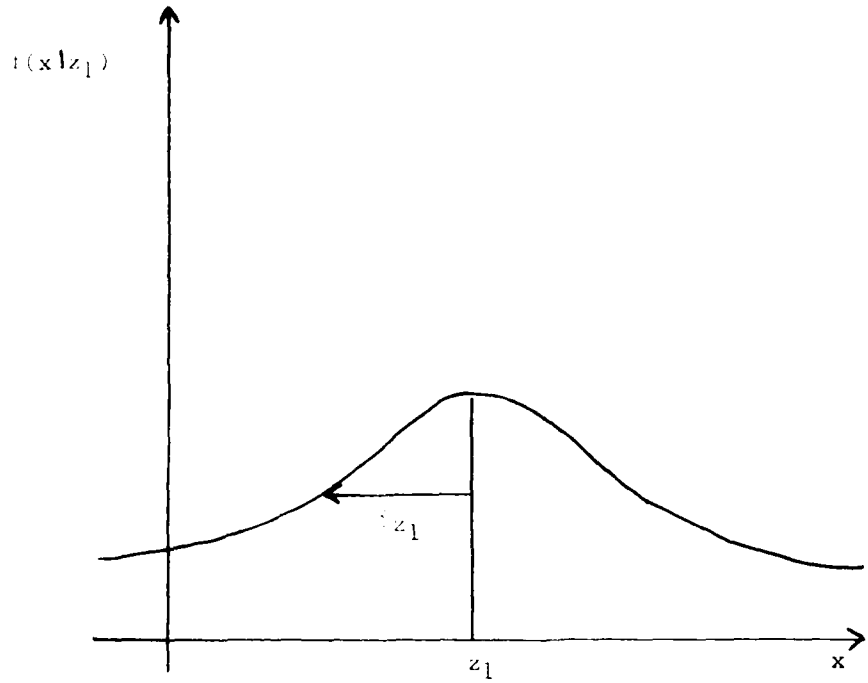


Figure 9. Conditional probability density of measurement z_1 alone. $f(x|z_1)$ is the probability that x is the actual value based on z_1.

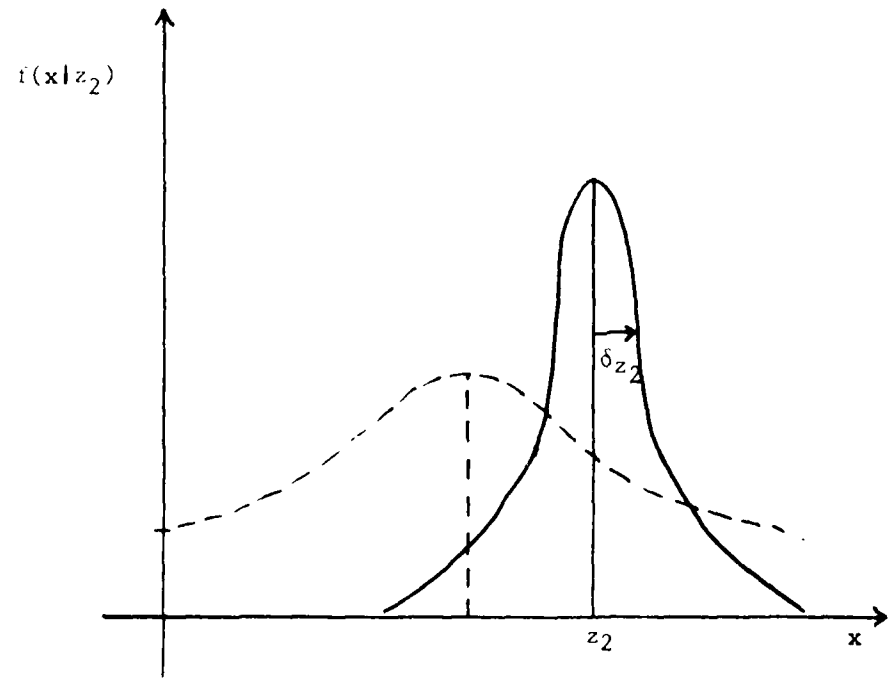


Figure 10. Conditional probability density based on position measurement z_2 alone.

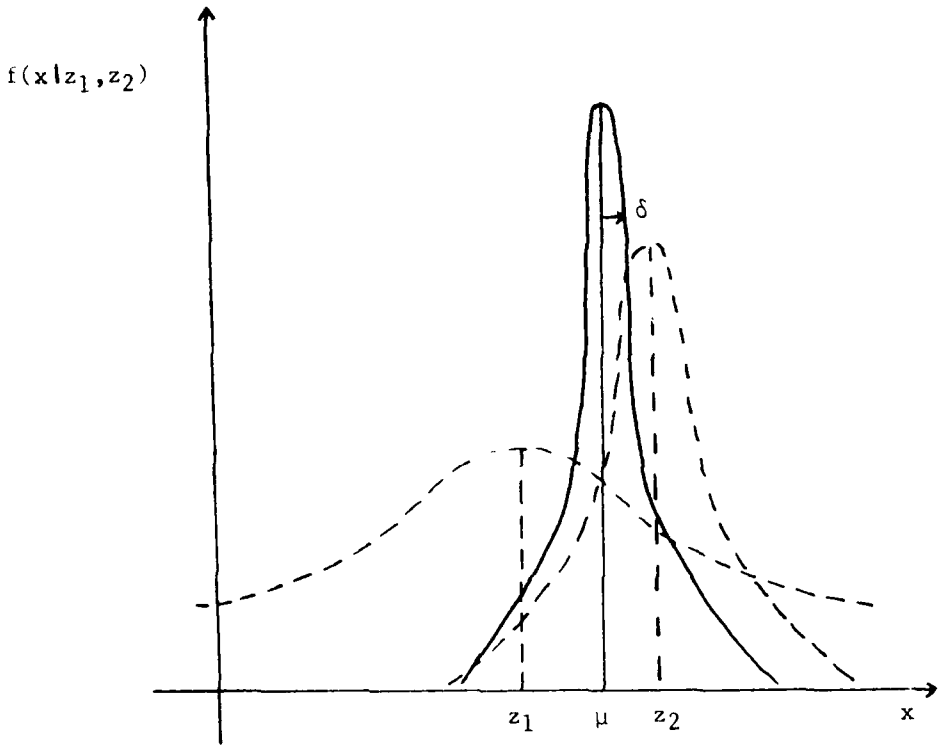


Figure 11. Conditional density of position based on measurements z_1 and z_2.

densities. It can also be shown to be the maximum likelihood estimate, the weighted least squares estimate, the minimum-variance unbiased linear estimate, and in every statistical sense, the "best" estimate of our position.

Equations (3) and (4) can be rewritten as

(5) $\hat{x}(t_2) = \hat{x}(t_1) + K(t_2)[z_2 - \hat{x}(t_1)]$

and

(6) $\sigma^2_x(t_2) = \sigma^2_x(t_1) - K(t_2)\,\sigma^2_x(t_1)$,

where

(7) $K(t_2) = \sigma^2_{z_1}/(\sigma^2_{z_1} + \sigma^2_{z_2})$.

So our estimate at time t_2 equals the best prediction of position before measurement z_2 plus a correction term of an optimal weighting times the difference between measurement z_2 and the prediction of its value. This difference is called the measurement residual, and the weighting factor $K(t_2)$ is the Kalman gain.

Without deriving the expressions above, examination of the filter behavior at extremes of measurement and prediction accuracy gives intuitive verification that it operates correctly. If the variances, or accuracies, of the two sightings are equal, our estimate gives the simple average of the two. If the second measurement is perfectly accurate, with a variance of zero, our estimate equals this second measurement. On the other hand, if the variance of our second measurement is infinitely large, that

measurement is discarded and our estimate equals the predicted value. Finally, even when one measurement is much worse than the other (but not infinitely bad), it reduces the variance of the combined estimate.

Now let's extend our example by adding motion, modeled by the differential equation:

(8) $dx/dt = u + w$

where u is the velocity of our boat and w is the noise, or uncertainty in the velocity, with variance σ_w^2.

Just before our next measurement at t_3, we obtain the following estimate by propogating our system model over time:

(9) $\hat{x}(t_3^-) = \hat{x}(t_2) + u[t_3 - t_2]$

and

(10) $\sigma_x^2(t_3^-) = \sigma_x^2(t_2) + \sigma_w^2[t_3 - t_2]$.

(Plus and minus superscripts indicate times just after and just before a measurement update, respectively.) The propogated values determine a conditional probability density describing our current estimated position. When we take a third measurement z_3 at time t_3, we once again combine the distribution of our estimate with that of our new measurement:

(11) $\hat{x}(t_3^+) = \hat{x}(t_3^-) + K(t_3^+)[z_3 - \hat{x}(t_3^-)]$

with

(12) $K(t_3^+) = \sigma_x^2(t_3^-)/(\sigma_x^2(t_3^-) + \sigma_{z_3}^2)$

and

$$\hat{x}(t_j^+) = \hat{x}(t_j^-) - K(t_j^+) \quad \hat{x}(t_j^-) \tag{13}$$

Again we observe the predictor-corrector format in which the residual is optimally weighted and used to update the state estimate.

To arrive at the general Kalman filter algorithm, we extend this example to the vector case and allow time-varying system parameters and noise descriptions.

The vector representations and matrix algebra operations used in the standard Kalman filter are a means of describing simultaneous operations on a number of different variables. The system state vector of a Kalman filter is a collection of all the variables in which we are interested. For example, an aircraft inertial navigation system might estimate the aircraft's altitude, latitude, longitude, pitch, yaw, direction and airspeed as a seven-element system state vector. This vector is updated by sensor measurements and propogated over time.

A few more ideas need to be introduced before looking at the standard Kalman equations. The first of these is the notion of a measurement observation matrix. Most measuring devices don't directly measure a single variable, but rather they measure a linear sum of all the variables, each scaled by some factor. We can show this mathematically with an m X n observation matrix where m

equals the number of measurements and n indicates the dimension of the system state vector. Again, n is just the number of variables in which we are interested. Each row of the matrix represents a single sensor measurement. Non-zero entries point out the underlying quantities contributing to the measurement, and the value of those non-zero entries is the scaling factor. Calling the collection of sensor measurements the vector $\underline{Z}$ (dimension m), and calling the system state vector $\underline{X}$, we model each set of sensor measurements with the following expression

$$(14) \quad \underline{Z}(t_i) = H(t_i)\,\underline{X}(t_i) + \underline{v}(t_i)$$

where $\underline{v}(t_i)$ equals the measurement noise (also of dimension m), and $H(t_i)$ is the measurement observation matrix.

Equation (14) does not appear explicitly in the Kalman filter, but the first term is used in calculating the measurement residual, and the second term helps to determine the Kalman gain.

Another needed concept is the covariance of a vector, or the expected error in that vector. When working with multiple variables, we have to consider not only each variable's uncertainty, but also the effect of that error on all the other variables. So system noises, measurement noises and the expected error of the system model itself are expressed as symmetric square matrices with the variances, or mean square errors, of each system variable on

the diagonal and the cross-correlations between errors making up the other terms. When the errors are initially uncorrelated, this covariance matrix is originally non-zero only on the diagonal. These initial values are determined by preliminary analysis and physical test.

Without going through the detailed derivation, the standard Kalman equations are given below as multi-dimensional extensions of the expressions in the preceding example. The Kalman gain matrix (n X m) is calculated in a manner similar to equation (12), where P represents the covariance matrix, or expected error, of the system model, R is the covariance matrix derived from the measurement noise $\underline{v}(t_i)$, and H is the measurement observation matrix. If we consider multiplication by an inverted matrix analagous to division, we can see the similarities between the two expressions:

(12) $$K(t_3^+) = \sigma_x^2(t_3^-)/(\sigma_x^2(t_3^-) + \sigma_{z_3}^2)$$

becomes

(15) $$K(t_3^+) = P(t_3^-)\, H^T(t_3)[H(t_3)\, P(t_3^-)\, H^T(t_3) + R(t_3)]^{-1}.$$

The covariance matrix is updated using the Kalman gain matrix like equation (13):

(13) $$\sigma_x^2(t_3^+) = \sigma_x^2(t_3^-) - K(t_3^+)\, \sigma_x^2(t_3^-)$$

becomes

(16) $$P(t_3^+) = P(t_3^-) - K(t_3^+)\, H(t_3)\, P(t_3^-).$$

And the final step in the measurement update is the

incorporation of the sensor data into the state vector:

(11) $\hat{x}(t_3^+) = \hat{x}(t_3^-) + K(t_3^+)[z_3 - \hat{x}(t_3^-)]$

becomes

(17) $\underline{\hat{X}}(t_3^+) = \underline{\hat{X}}(t_3^-) + K(t_3^+)[\underline{\tilde{Z}}(t_3) - H(t_3)\ \underline{\hat{X}}(t_3^-)]$

where $\underline{Z}(t\ )$ represents the actual measurements.

Lastly, system dynamics must be modeled to allow time propagation between measurement updates. The system model looks very much like the example:

(8) $dx/dt = u + w$

becomes

(18) $\dot{\underline{X}}(t_i) = F(t_i)\ \underline{\hat{X}}(t_i) + \underline{w}(t_i)$

where $F(t_i)$ is a transition matrix describing how the state vector changes over time and $\underline{w}(t_i)$ represents system noise. The actual time propagation separates out the noise term just as in our example. A covariance matrix (Q below) representing the growth in the expected error of $\underline{\hat{X}}(t_i)$ due to system noise is derived from $\underline{w}(t_i)$, and included in the expression which propagates the system error covariance. Equation (19) functionally corresponds to the error dynamics expressed in our example problem's equation (10):

(19) $\dot{P}(t_i) = F(t_i)\ P(t_i) + P(t_i)\ F^T(t_i) + Q(t_i)$.

Thus the change in system error covariance over time is modeled by applying the state transition matrix to the rows and the columns of the covariance matrix and adding

in the expected growth in system error.

The five equations (15) - (19) make up the standard Kalman filter. The latter two (less the noise term in (18)) are numerically integrated over time to give our predicted system values and expected errors. Equations (15), (16), and (17) incorporate the actual measurements into the filter estimates.

This standard Kalman filter is prone to serious numerical difficulties because it requires the subtraction of numbers of very different magnitude. We can solve this by using double precision arithmetic, at a cost of time and hardware. A more elegant solution performs the measurement update computations on the square root of the covariance matrix, and processes each of the measurements separately as scalars rather than simultaneously as a vector. These "square root" filters are algebraically equivalent to the standard Kalman filter, but give twice the effective precision with the same computer word length. This project used the form of the square root filter developed by Dr. Neal A. Carlson. His formulation determines the covariance square root matrix in a triangular form and keeps it that way during update, resulting in reduced memory and computational requirements. Furthermore, by processing measurements singly we avoid the inversion of anything larger than a 1 X 1 matrix. So we save the

storage space required by a matrix inversion routine.

The measurement update computations are given below, where m is the number of sensor measurements taken at each update and n is the dimension of the state vector. "$\sqrt[c]{\ }$" stands for the Choleski decomposition of a matrix.

$$\underline{S}^{-} = \sqrt[c]{\underline{P}^{-}}$$

For $j = 1$ to m

$$\underline{d} = (\underline{S}^{-})^{T} H_j^{T}$$

$$A_0 = R_j$$

$$\underline{b} = \underline{0}$$

For $i = 1$ to n

$$A_i = A_{i-1} + d_i^2$$

$$a_i = (A_{i-1}/A_i)^{1/2}$$

$$c_i = d_i/(A_{i-1} A_i)^{1/2}$$

$$\underline{b}_i = \underline{b}_{i-1} + \underline{S}_i^{-} d_i$$

$$\underline{S}_i^{+} = \underline{S}_i^{-} a_i - \underline{b}_{i-1} c_i$$

End for

$$\hat{\underline{X}}^{+} = \hat{\underline{X}}^{-} + (\underline{b}_n / A_n) \tilde{DZ}_j$$

End for

Where H_j = jth row of the measurement observation matrix,

$R_j = R_{jj}$ element of measurement noise covariance matrix,

d_i = ith element of $\underline{d}$,

$\underline{S}_i^{-}$ = ith column of $\underline{S}^{-}$,

$\tilde{DZ}_j = \tilde{Z}_j - H_j \hat{\underline{X}}^{-}$, or the measurement residual.

When these calculations are done, the updated covariance matrix is formed by squaring its root ($P = SS^T$) and both it and the system state vector are propagated to the next measurement update.

A final noteworthy characteristic of the filter used for this project is its feedback operation. Rather than tracking the actual position and velocity, the filter propagates the errors in the Inertial Navigation System's (INS) determination of those quantities. In each measurement update, not only are these propagated error estimates used to correct the INS readings, but the internal variables of the INS device itself are corrected to incorporate the filter information. After this feedback, the INS device is error-free and the filter is reset to zero to reflect that fact.

SECTION IV

CODE OPTIMIZATION ON ARRAY MACHINES

1. ALGORITHM DESIGN

The first strategy for writing efficient vector processor code is a global one: restructure the algorithm to match the architecture of the machine. Generally this means reformulating scalar processing into vector operations. An example for the Carlson square root filter of this project is described below.

The following equations were introduced in Section 3 and form part of the filter covariance measurement update as commonly expressed in the Carlson algorithm (n is the dimension of the filter, underlined quantities are n-element vectors). Scalar and vector operations are labeled accordingly.

$A_0 = R_j$

For i = 1 to n

$$A_i = A_{i-1} + d_i^2 \qquad (1) \text{ scalar}$$

$$a_i = (A_{i-1}/A_i)^{1/2} \qquad (2) \text{ scalar}$$

$$c_i = d_i/(A_{i-1} A_i)^{1/2} \qquad (3) \text{ scalar}$$

$$\underline{b}_i = \underline{b}_i + \underline{S}_i d_i \qquad (4) \text{ vector}$$

$$\underline{S}^{+}_{i} = \underline{S}^{-}_{i} a_{i} - \underline{b}_{i-1} c_{i} \qquad (5) \text{ vector}$$

End for

The recursive nature of equation (1) resists vectorization, however, if vector A of dimension n+1 is calculated in a preliminary step, then its square root can be obtained in a vector operation and equations (2) and (3) become operations on successive elements of a fully determined vector, and they can be streamed through the array hardware. This provides three more opportunities to take advantage of the greater efficiency of vector operations over scalar ones. Equations (4) and (5) remain unchanged.

The resulting steps are shown below, labeled as scalar or vector operations.

$A_0 = R$

For i = 1 to n

$$A_i = A_{i-1} + d_i^2 \qquad (1) \text{ scalar}$$

End for

For i = 1 to n

$$\underline{A'}_{i} = (\underline{A}_{i}) \qquad (1a) \text{vector}$$

$$\underline{a}_{i} = \underline{A'}_{i-1} / \underline{A'}_{i} \qquad (2) \text{ vector}$$

$$\underline{c}_{i} = \underline{d}_{i} / (\underline{A'}_{i-1} \underline{A'}_{i}) \qquad (3) \text{ vector}$$

$$\underline{b}_{i} = \underline{b}_{i-1} + \underline{S}^{-}_{i} \underline{d}_{i} \qquad (4) \text{ vector}$$

$$\underline{S}^{+}_{i} = \underline{S}^{-}_{i} a_{i} - \underline{b}_{i-1} \underline{c}_{i} \qquad (5) \text{ vector}$$

End for

2. GENERAL CODING ISSUES

Once we have completely vectorized our algorithm, some fairly obvious general coding issues arise.

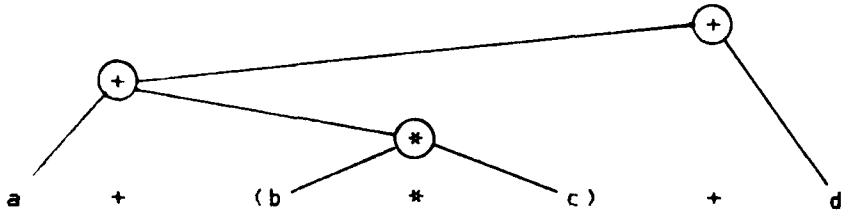
Data storage is a pivotal point, since data flow is one of the major obstacles to achieving optimal parallelism. Arrays stored in main memory must be so located as to minimize interleave violations. If a matrix is operated on by columns, that is how it should be stored so elements fetched successively are in different memory banks.

Within the CPU, cache register storage must be utilized to eliminate the bottleneck of main memory access. In the AP-120B, this means storage of operands and results in both data pads and table memory RAM if available. Vector dimensions, increments and memory locations can be stored in the 16-bit registers. Besides fetching arrays into the data pads prior to operations on them, it makes sense to store frequently accessed variables permanently in these registers.

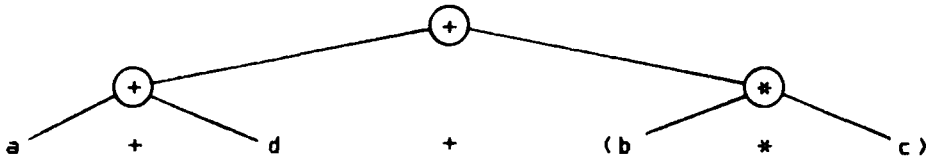
Another obvious way to take advantage of functional parallelism is simply to maximize the concurrency of independent operations. Computations are overlapped with the storage of a previous operation's results, or fetches of operands for the next calculation. Integer arithmetic is done simultaneously with floating-point operations to per-

form loop control and address calculation.

Lastly, arithmetic expressions written with conventional computers in mind need to be restructured for parallel machines. Expression tree height reduction can reduce the time necessary to evaluate a complex expression. For example:



the tree shown above has three levels indicating data dependence for successive calculations. It would require seven CP cycles to evaluate on the AP left-to-right as written. If rearranged into an expression of height two:



we uncover parallelism and allow the calculation to be made in five CP cycles.

3. LOOP OPTIMIZATION

By far the most important consideration in optimizing code for vector algorithms is minimization of loop length. One means of achieving this, as presented by Floating Point Systems Inc. in their manuals and programming course, first requires a determination of hardware resources for the loop computations. Once we know how many multiplies, adds, memory accesses and integer calculations are in the loop, it is possible to calculate the minimum loop length. Serial code to do the calculation is then written, and "wrapped" into columns, each of minimum length. For setup and cleanup, this code is "unwrapped" by columns before and after the loop.

Explanation requires an example. We will exercise the technique on a small loop that updates the Kalman filter using the recomputed gain array (G) and the measurement residual (ZRES). The FORTRAN code looks like this:

```
      Do 1 I = 1,n
    1 X(I) = X(I) + G(I) * ZRES
```

where n is the dimension of both the filter state vector and the gain array.

To implement a serial version of this loop in AP Assembly Language, we have to first set up the computation. This involves loading the array length into an integer register labeled N, setting the Data Pad Index Register to the location of the filter state vector, and fetching ZRES from MD memory into DPY(0). Last we load the address of array G into another integer register. We store the state vector in the Data Pad permanently to eliminate memory fetches. Once these operations are complete, our loop is as shown in Figure 12. Semi-colons in AP Assembly Language code are used to separate the discrete operations of a single micro-instruction, operations which will occur simultaneously.

The first step in optimizing this loop is a determination of hardware constraints on loop size. Instructions 1 and 12 both use the integer ALU, so that forces a minimum length of two. No other resources are used more than once, so two is the theoretical minimum we are seeking. Now we simply re-write the linear code in as many columns as necessary to form a loop two instructions in length, making sure that the branch is in the second row/instruction for obvious reasons. If we run into a hardware conflict in one of the rows such as simultaneous S-Pad operations, we just insert a NOP and put our conflicting instruction in the next row/column location.

```
1.   LOOP:    INC G; SETMA      "Fetch G(I)
2.            NOP               "Await MD access
3.            NOP
4.            FMUL DPY,MD       "G(I)*ZRES
5.            FMUL              "Pipeline pusher
6.            FMUL              "Pipeline pusher
7.            FADD FM,DPX       "[G(I)*ZRES] + X(I)
8.            FADD              "Pipeline pusher
9.            DPX<FA            "Store new X(I)
10.           INCDPA            "Increment Data Pad Index
11.           DPY<DPY(-1)       "Move ZRES to DPY(Current Index)
12.           DEC N             "Loop Control
13.           BGT LOOP          "Branch until done
```

Figure 12. Linear code for filter update.


```
LOOP: INC G;SETMA ;{NOP}        ;FMUL     ;FADD FM,DPX   ;DPX<FA    ;DPY<DPY(-1)   ;{NOP}
      {NOP}       ;FMUL DPY,MD ;{FMUL}   ;FADD          ;INCDPA    ;DEC N         ;BGT LOOP
```

Figure 13. "Wrapped" loop code. Op-codes enclosed in braces are left for documentation purposes.

```
LOOP: {NOP}       ;{NOP}     ;FMUL DPY,MD  ;{FMUL}        ;FADD      ;INCDPA       ;DEC N
      INC G;SETMA ;{NOP}     ;FMUL         ;FADD FM,DPX   ;DPX<FA    ;DPY<DPY(-1)  ;BGT LOOP
```

Figure 14. Corrected loop code

Each column represents operations on successive elements of the array. The results are shown in Figure 13.

As a first step, this looks okay, but it won't work. Our loop decrement is done in the second row where the branch is done, but integer condition codes are not set until one cycle after the operation. This means that we would be branching on the results of the G address calculation in column one. Let's move our fetch down to row two so the decrement can go in row one, giving us the code of Figure 14.

This loop will work. All that remains is the appropriate setup and cleanup, and a close examination of index values in the Data Pad. This last check reveals that in the two-instruction loop with its extensive setup, we are storing into DPX after we have incremented its index. We should store into DPX one behind the updated index. With this change, the complete loop is shown in Figure 15. As you can see, setup and cleanup involve propagating all columns both up-to-the-right to fill our pipeline and down-to-the-left to empty it.

Our original loop had 13 instructions. The optimized loop is 2 instructions long with a setup and cleanup overhead of 15 instructions. For a sample case where n is equal to 100, the linear loop executes in:

```
"********** LOOP SETUP **********
                                                                    INC G;SETMA
                                                        NOP         ;{NOP}
                                                        INC G;SETMA ;{NOP}
                                          {NOP}         ;{NOP}      ;FMUL DPY,MD
                                          INC G;SETMA   ;{NOP}      ;FMUL
                                {NOP}     ;{NOP}        ;FMUL DPY,MD ;{FMUL}
                                INC G;SETMA ;{NOP}      ;FMUL        ;FADD FM,DPX
"********** BEGIN LOOP **********
LOOP: {NOP}       ;{NOP}      ;FMUL DPY,MD ;{FMUL}     ;FADD        ;INCDPA      ;DEC N
      INC G;SETMA ;{NOP}      ;FMUL        ;FADD FM,DPX ;DPX(-1)<FA  ;DPY<DPY(-1) ;BGT LOOP
"********** END LOOP **********
"********** LOOP CLEANUP **********
      {NOP}       ;FMUL DPY,MD ;{FMUL}     ;FADD        ,INCDPA
      {NOP}       ;FMUL        ,FADD FM,DPX ;DPX(-1)<FA  ;DPY<DPY(-1)
      FMUL DPY,MD ;{FMUL}      ,FADD        ;INCDPA
      FMUL        ;FADD FM,DPX ;DPX(-1)<FA
      FMUL        ;FADD        ,INCDPA
      FADD FM,DPX ;DPX(-1)<FA
      FADD
      DPX<FA
"********** END LOOP CLEANUP **********
```

Figure 15. Complete optimized 2-instruction loop with setup and cleanup

100 loops * 13 cycles/loop = 1300 cycles = 216.67 μs

The optimized loop requires:

15 cycles setup + (96 loops * 2 cycles/loop)

= 207 cycles = 34.50 μs.

Only 96 loops are required with the optimized code since 4 iterations take place in the setup and cleanup. This loop optimization netted a speedup of 6.28 times for a 100 element array. Even for the 5-element array of our sample filter the speedup factor is 3.82.

Not all loop optimizations are as straight forward as the one just examined, although they are all best approached using the same column-wrapping technique. A calculation from the numerical integration serves to illustrate a single-instruction loop with a register access problem and delayed loop count response. Figure 16 shows the APAL statements, with the in-line documentation. Tightly-wrapped loops are by nature obscure. With such elaborate setup and cleanup, and the simultaneous operations on different vector elements, it is essential to include prolific comments in the code.

Calculation of the 1st order integration result in

```
    LDMA;    DB=MDSCR                          "FETCH H3 INTO MD REGISTER
    LDSPI CTR;       DB=N-5                    "LOOP COUNT
    LDDPA;   DB=Y+3                            "DP INDEX = Y(4)
"********** LOOP SETUP **********
    FMUL DPY(-3),MD                            "PY(1)*H3
    FMUL DPY(-2),MD                            "PY(2)*H3
    FMUL DPY(-1),MD                            "PY(3)*H3
    FMUL DPY,MD;                               "PY(4)*H3
    FADD FM,DPX(-3)                            "[PY(1)*H3]+Y(1)
    FMUL DPY(1),MD;                            "PY(5)*H3
    FADD FM,DPX(-2);                           "[PY(2)*H3]+Y(2)
    DEC CTR                                    "LOOP CONTROL - 1 CYCLE LAG
"********** START K4A1 LOOP **********
K4A1:       FMUL DPY(2),MD;                                "PY(I+2)*H3
    FADD FM,DPX(-1);                           "[PY(I-1)*H3]+Y(I-1)
    DPX(-3)<FA;                                "STORE NEW Y(I-3)
    INCDPA;                                    "I=I+1
    DEC CTR;                                   "LOOP CONTROL
    BGT K4A1
"********** END K4A1 LOOP **********
"
"********** LOOP CLEANUP **********
    FMUL;                                      "PIPELINE PUSHER
    FADD FM,DPX(-1);                           "[PY(N-2)*H3]+Y(N-2)
    DPX(-3)<FA                                 "STORE NEW Y(N-4)
    FMUL;                                      "PIPELINE PUSHER
    FADD FM,DPX;                               "[PY(N-1)*H3]+Y(N-1)
    DPX(-2)<FA                                 "STORE NEW Y(N-3)
    FADD FM,DPX(1),                            "[PY(N)*H3]+Y(N)
    DPX(-1)<FA                                 "STORE NEW Y(N-2)
    FADD;                                      "PIPELINE PUSHER
    DPX<FA                                     "STORE NEW Y(N-1)
    DPX(1)<FA                                  "STORE NEW Y(N)
"********** END LOOP CLEANUP **********
```

Figure 16. Single-Instruction Optimized Loop

KUTMER satisfies the expression:

```
      DO  10 J = 1,N
   10 Y(J) = Y(J) + H3*PY(J)
```

where H3 is one third of the integration step size, and PY(J) is the derivative of Y(J). Y is the concatenation of the filter state vector and the upper triangular vector representation of the covariance matrix. Y is stored permanently in DPX registers 12 through 31, and PY in the same area of DPY. This results in the problem of where to keep H3 during the computation.

H3 has been previously calculated and stored in DPX(7). Data Pad access is controlled by an index register with a value in the range 0 to 31. In a single instruction, eight registers in each Data Pad are accessible by specifying, in effect, an index to this primary index. The secondary index can range from -4 to +3. Hence, DPX(7) is unreachable when operating on Y and PY. The solution chosen was to write H3 to a scratch area in MD memory, fetch it, and leave it in the memory input buffer throughout the loop. This buffer register communicates with the floating multiplier unit via one of the two multiplier buses, and as long as no other fetches are done, its value is unchanged.

Another idiosyncracy of this loop is a single counter decrement that takes place one instruction before beginning the loop. This is necessary because integer condition codes are not set until the instruction following the operation. Therefore, in a single-instruction loop the control index has to reach zero one iteration before the calculations are complete.

A word about the documentation. In the loop, "I" represents the current Data Pad index with respect to Y and PY. This index points to Y(4) for the first iteration, and Y(19) for the last. "N" is equal to the dimension of Y, which is twenty in the filter of this project, hence Y(N) is the last element of vector Y. The unoptimized code for this loop is 13 instructions long, while the code of Figure 16 is only 1. With a setup and cleanup overhead of 10 instructions, the optimized loop executes in 10 + n CP cycles where n is the dimension of Y. The linear code requires 13*n CP cycles.

As a final example of code optimization, a nested loop structure from the measurement update will be examined. The FORTRAN code is shown in Figure 17. The problem with optimizing this calculation lies in the iteration count of the inner loop, which is equal to the current loop index of the outer loop. This means that in our first execution of the outer loop, the nested loop is exe-

cuted only once, and so on. Optimized loops require setup to fill the functional unit pipelines and cleanup to empty them, and this particular inner loop completes calculations on four elements of the vector operands in a single pass through setup, the loop itself, and cleanup. In other words, once through the loop performs four iterations. This is fine when the iteration count is four or greater, but poses a problem when it is less.

```
      K = 0
      DO 20 I=1,N
          STHT(I) = 0.
          DO 20 J=1,I
              K = K + 1
              STHT(I) = STHT(I) + S(K)*H(J)
20    CONTINUE
```

Figure 17. Nested FORTRAN Loop

No harm is done by fetching and operating on vector elements beyond the intended iteration range, as long as 1) the S address pointer which is over-incremented in the inner loop is reset back to its correct value for the next iteration and 2) no unintended results are stored into a result register or MD memory. By running once through the setup and the loop itself, the first result has been stored, and we are in various stages of completion on the next three calculations. It is in loop cleanup that unwanted results are stored when the inner loop count is less than four.

Once the problem is located, a solution is obvious. A second counter is used with the intended iteration count. Unlike the loop control counter, it is not set to three less than the iteration count to compensate for setup and cleanup. This second counter is decremented separately in the cleanup, enabling branches out of the cleanup before unintended results are stored.

To assure that the S address pointer is correct, an updated copy of it is kept and used in the outer loop to reset the pointer which has been over-incremented in the inner loop. The correct optimized code follows in Figure 18. The unoptimized code for the calculation has 11 instructions in its outer loop, the last of which is not executed on the final iteration. Thus 11n-1 instructions are executed for a filter of dimension n. Inner loops which iterate from 1 to the outer loop index are executed n(n + 1)/2 times, where n is the number of outer loop iterations. The unoptimized inner loop code is made of 12 instructions, so the expression to give us the total number of instructions executed to fully iterate outer and inner loops is

$$[12*(n(n + 1)/2) + (11n - 1)].$$

The optimized loop is trickier to time. There are 14

```
"
"    COMPUTE STHT=(H*S)-TRANSPOSE
"            K=0
"            DO 20 I=1,N
"                    STHT(I)=0.
"                    DO 20 J=1,I
"                            K=K+1
"    20      STHT(I)=STHT(I)+S(K)*H(J)
"
XSPLUS:      LDSPI SADRHLD;   DB=S
     LDSPI STHTADR;   DB=STHT
     LDDPA;   DB=5;                          "SET DATA PAD INDEX TO FIVE
     CLR J
     LDSPI N,         DB=DIM                 "SET LOOP COUNT
XS1: DPX<ZERO;                               "DPX=STHT(I)=ZERO
     ADD J,SADRHLD                           "SET LOCATION IN S FOR INNER LOOP
     MOV SADRHLD,SADR
     INC J                                   "SET INNER LOOP COUNT
     MOV J,K
     LDSPI UTILA;     DB=3                   "SET SPAD UTILA TO 3
     SUB UTILA,K                             "DECREMENT LOOP COUNT BY 3 FOR SETUP
     MOV J,K2                                "COUNTER TO JUMP OUT OF LOOP
     DEC  UPSW                               "CHECK WHICH UPDATE
     BLT HUP1                                "UPSW=0 MEANS 1ST UPDATE
     LDSPI HADR;      DB=H2;                 "UPSW=1 MEANS 2ND UPDATE
     BR XS2SET
HUP1:LDSPI HADR;      DB=H1
```

Figure 18. Optimized Inner Loop

```
"********** LOOP SETUP **********
XS2SET: INC SADR;      SETMA                    "FETCH S(K)
     NOP                                        "AWAIT MD ACCESS
     INC HADR;         SETMA                    "FETCH H(1)
     DPY<MD;                                    "STORE S(K)
     INC SADR;         SETMA                    "FETCH S(K=K+1)
     NOP                                        "AWAIT MD ACCESS
     FMUL DPY,MD;                               "S(K-1)*H(1)
     INC HADR;         SETMA                    "FETCH H(2)
     FMUL;                                      "PIPELINE PUSHER
     DPY<MD;                                    "STORE S(K)
     INC SADR;         SETMA                    "FETCH S(K=K+1)
     FMUL                                       "PIPELINE PUSHER
     FADD FM,DPX;                               "STHT(I)+[S(K-2)*H(1)]
     FMUL DPY,MD;                               "S(K-1)*H(2)
     INC HADR;         SETMA                    "FETCH H(3)
"********** START XS2 LOOP **********
XS2: FADD;                                      "PIPELINE PUSHER
     FMUL;                                      "PIPELINE PUSHER
     DPY<MD;                                    "STORE S(CURRENT K)
     INC SADR;         SETMA                    "FETCH S(NEXT K)
     DEC K;                                     "LOOP CONTROL
     DPX<FA;                                    "STORE STHT(I)
     FMUL                                       "PIPELINE PUSHER
     BGT XS2;                                   "BRANCH FOR INNER LOOP
     FADD FM,DPX;                               "STHT(I)+[S(PREVIOUS K)*H(PREVIOUS J)]
     FMUL DPY,MD;                               "S(CURRENT K)*H(CURRENT J)
     INC HADR;         SETMA                    "FETCH H(NEXT J)
"********** END XS2 LOOP **********
"
```

Figure 18. Continued

```
"********** LOOP CLEANUP **********
    DEC K2;                                  "CLEANUP LOOP CONTROL
    FADD;                                    "PIPELINE PUSHER
    FMUL;                                    "PIPELINE PUSHER
    DPY<MD                                   "STORE S(LAST K)
    BEQ XS2END;       BLT XS2END             "INHIBIT STORES IF J<2
    DPX<FA;                                  "STORE STHT(I) -> J-2 ITERATION
    FMUL                                     "PIPELINE PUSHER
    DEC K2;                                  "CLEANUP LOOP CONTROL
    FADD FM,DPX;                             "STHT(I)+[S(2ND LAST K)*H(2ND LAST J)]
    FMUL DPY,MD                              "S(LAST K)*H(LAST J)
    BEQ XS2END;       BLT XS2END;            "INHIBIT STORES IF J<3
    FADD;                                    "PIPELINE PUSHERS
    FMUL
    DPX<FA;                                  "STORE STHT(I) - J-1 ITERATION
    FMUL                                     "PIPELINE PUSHER
    DEC K2;                                  "CLEANUP CONTROL
    FADD FM,DPX                              "STHT(I)+[S(LAST K)*H(LAST J)]
    BEQ XS2END;       BLT XS2END;            "INHIBIT STORES IF J<4
    FADD                                     "PIPELINE PUSHER
    DPX<FA                                   "STORE STHT(I) - JTH ITERATION
"********** END LOOP CLEANUP **********
XS2END: INC STHTADR; SETMA;   MI<DPX          "WRITE STHT(I) TO MD
    DEC N
    BLT XS1END;       BEQ XS1END             "THESE TWO LINES OF CODE ARE
    JMP XS1                                  "EQUIVALENT TO 'BGT XS1'
```

Figure 18. Continued

outer loop instructions, and again the final instruction is not executed on the last iteration. An instruction count expression for the three-instruction optimized inner loop was derived (see Section 6) to be 51 + (n - 3)21 + [(n - 3)(n - 4)/2]*3. The overall expression is:

$$(14n - 1) + [51 + (n - 3)21 + [(n - 3)(n - 4)/2]*3].$$

These expressions give us an instruction count of 234 for the unoptimized loop with a filter dimension of five, while the optimized loop requires only 165 CP cycles for the same filter.

The improvement becomes much more dramatic for a 20-element filter vector. Here the unoptimized code requires 2739 cycles and the "wrapped" loop code executes in 1452 instruction cycles, almost a 2:1 speedup.

Another way to reduce loop size that was not seen above is the reallocation of functional units. For example, integer address calculation and index decrements often account for the highest single resource usage. If the floating adder unit has free cycles, we can use it to control loop iteration, reducing S-Pad ALU usage and loop size by one instruction.

Three major complications are introduced by this scheme. First, it becomes necessary to store a

floating-point 1.0 where it will be handy for index decrements. The Table Memory buffer register is generally used for this, since ROM Table Memory is the most convenient source of a floating-point one. Secondly, condition codes for adder operations are not set until two cycles after the operation, which means loops shorter than three instructions are branching on loop counts from previous iterations. Third, if a loop includes more than one other floating adder operation, it will not be possible to simply cycle the loop count through the functional unit. The count will have to be temporarily stored in a Data Pad register. Obviously, these additional complications must be handled within the reduced loop size in order for the floating point loop count to increase execution speed.

Unlike the examples above, it is not always possible to achieve the minimum loop size. Involved computations requiring the temporary storage of intermediate results generally encounter a register access bottleneck. In this case, it can be helpful to rearrange the sequence of calculations, minimizing data pad usage by streaming intermediate results directly into another functional unit. Also, constants and intermediate results can be "stored" in the floating point units with a zero add or identity multiply. Sometimes, however, the only solution is to increase the number of instructions in the loop, allowing

for more Data Pad access instructions.

4. CONCLUSION

The software techniques discussed in this section cover the gamut of available means to optimize code for a parallel, pipelined array processor. Their application demands a sophistication beyond that of current compilers, so optimal code for these architectures must be painstakingly hand-tooled. In Sections 6 and 7, we will look at the kind of results this hand-coding achieved, both for small portions of highly vectorized routines, and for a complete filter implementation including control, output, and a large number of strictly integer operations.

SECTION V

PROGRAM DOCUMENTATION

Essentially a Kalman filter does three things: output filter estimates, update those estimates with sensor measurements, and propagate the filter via numerical integration. The AP code is divided into three overlay segments to accomplish these functions. The root segment consists of the initialization, control and output routines (VECTRN, APCTL, OUT) which are always resident in PS memory. Overlay segment two does the numerical integration (KUTMER, DERIV, FPPPFT), and segment three performs the measurement update (PSQRT, UPDATE, XSPLUS, SQRS). The control routine simply propagates the filter to the next scheduled output or measurement update, swapping overlay segments two and three before and after the updates. Figure 19 gives a pseudo-code description of the filter algorithm implemented in the AP-120B. Subroutine names shown on the left specifically locate the functions listed opposite them. XF represents the filter state vector, PF is the covariance matrix, T is the current time, and H is the integration step size.

A switch normally off can be set to generate output during the filter update stage. APCTL calls subroutine

OUT four times during the update, and if the switch is off, control returns with nothing written.

```
APCTL       Initialize
VECTRN
APCTL       Load Integration Overlay
          ┌LOOP FOREVER
APCTL     │      TUP = T + H
KUTMER    │      Propogate XF and PF from T to TUP
DERIV     │
FPPPFT    │
APCTL     │    ┌IF T = Output
OUT       │    │THEN Write XF, PF to MD
          │    └ENDIF
APCTL     │    ┌IF T = Update
APCTL     │    │THEN   Load Update Overlay
PSQRT     │    │       S = Decomposition of PF
UPDATE    │    │       Read first measurement and
          │    │          calculate residual
XSPLUS    │    │       Update S and XF
UPDATE    │    │       Read second measurement and
          │    │          calculate residual
XSPLUS    │    │       Update S and XF
SQRS      │    │       PF = S * S-Transpose
APCTL     │    │       Reset XF = 0
APCTL     │    │       Load Integration Overlay
          │    └ENDIF
APCTL     │     IF T = Final Time
OUT       │     THEN Write F, PF to MD
          │          STOP
          │     ENDIF
          └END LOOP
```

Figure 19. Pseudo-Code Program Description

The covariance matrix is a 5 x 5 symmetric matrix, but it is an upper triangular vector representation. This eliminates duplicate computations and saves filter storage space. To propagate the filter state vector and the covariance matrix, a vector labelled Y of length twenty is

formed by concatenating XF and PF, and numerically integrating it from T to TUP in steps of length H. With PY(T,Y) representing the derivative of Y at time T, the Kutta-Merson fifth-order integration equations that subroutine KUTMER implements are:

$$
\begin{aligned}
Y0 &= Y(T) \\
Y1 &= Y0 + (H/3)PY(T, Y0) \\
Y2 &= Y0 + (H/6)PY(T, Y0) + (H/6)PY(T + H/3, Y1) \\
Y3 &= Y0 + (H/8)PY(T, Y0) + (3H/8)PY(T + H/3, Y2) \\
Y4 &= Y0 + (H/2)PY(T, Y0) - (3H/2)PY(T + H/3, Y2) \\
&\quad + (2H)PY(T + H/2, Y3) \\
Y5 &= Y0 + (H/6)PY(T, Y0) + (2H/3)PY(T + H/2, Y3) \\
&\quad + (H/6)PY(T + H, Y4) \\
&= Y(T + H)
\end{aligned}
$$

KUTMER calls DERIV to calculate the derivative of Y at each required time interval. These derivatives are formed as follows:

```
XFDOT(1) = XF(2)
XFDOT(2) = -XF(3)*6 + XF(5)
XFDOT(3) =  XF(2)/RE + XF(4)
XFDOT(4) =  -XF(4)/TAUF(1)
XFDOT(5) =  -XF(5)/TAUF(2)
PFDOT    = P*F + P*F-Transpose + QF
```

The first two terms of PFDOT are calculated in subroutine FPPPFT and DERIV adds in the non-zero QF values.

The 11/70 FORTRAN code reads initial values for filter variables and constants, prints those values, and transfers them to AP MD memory. It then loads the AP object module and transfers control to the AP until the particular filter scenario being run has completed. Lastly, the 11/70 program fetches all the filter output from a file in AP MD memory and prints it.

The assembly language code itself is generously commented. APAL routines include FORTRAN expressions of the algorithms they implement as well as notes explaining every line of assembly language code.

Detailed program documentation is in HIPO format, divided into two sections that correspond to the 11/70 FORTRAN program and the AP-120B routines. Each section contains a hierarchical chart defining program control and

structure followed by individual subroutine descriptions. The latter show inputs on the left, outputs on the right, and processes in the middle. Preceding these two sections is a table describing all variables and constants used in the filter.

1. VARIABLE DESCRIPTIONS

FILTER VARIABLES:

A(9)	Work Area
ALPHA	ZRES Standard Deviation
DEWK(80)	Work Area
DTMEAS	Measurement Intervals
DTPRNT	Output Intervals
F(7)	State Vector Transition Matrix Non-Zeroes
G	Acceleration of Gravity
G(5)	Kalman Gain Vector
H(5)	Measurement Observation Matrix Row
HO	Integration Interval
IMEAS	Update Measurement Number
INDF(2,7)	Location of F Matrix Non-Zero Elements
INDQ(2,2)	Location of QF Matrix Non-Zero Elements
M	Number of Measurements per Update = 2
NALL	State/Covariance Dimension = 20
NF	Filter State Vector Dimension = 5
NTR	Size of Triangular Part of PF = 15
NZF	Number of Non-Zeroes in F = 7
NZQ	Number of Non-Zeroes in QF = 2
PF(15)	Filter Covariance Matrix
PFO(15)	Initial Filter Covariance Matrix
PFDOT(15)	Derivative of Filter Covariance Matrix
QF(2)	Noise Error Covariance Matrix Non-Zeroes
RE	Earth's Radius
RFVCTR(2)	Measurement Noise Variances
S(15)	Cholesky Decomposition of PF
SDWF(2)	White Noise Standard Deviations
T	Current Filter Time
TO	Initial Run Time
TAUF(2)	Time Constants
TF	Final Run Time
TMEASO	Initial Measurement Update Time
XF(5)	Filter State Vector
XFO(5)	Initial Filter State Vector
XFDOT(5)	Derivative of Filter State Vector
ZRES	Measurement Residual

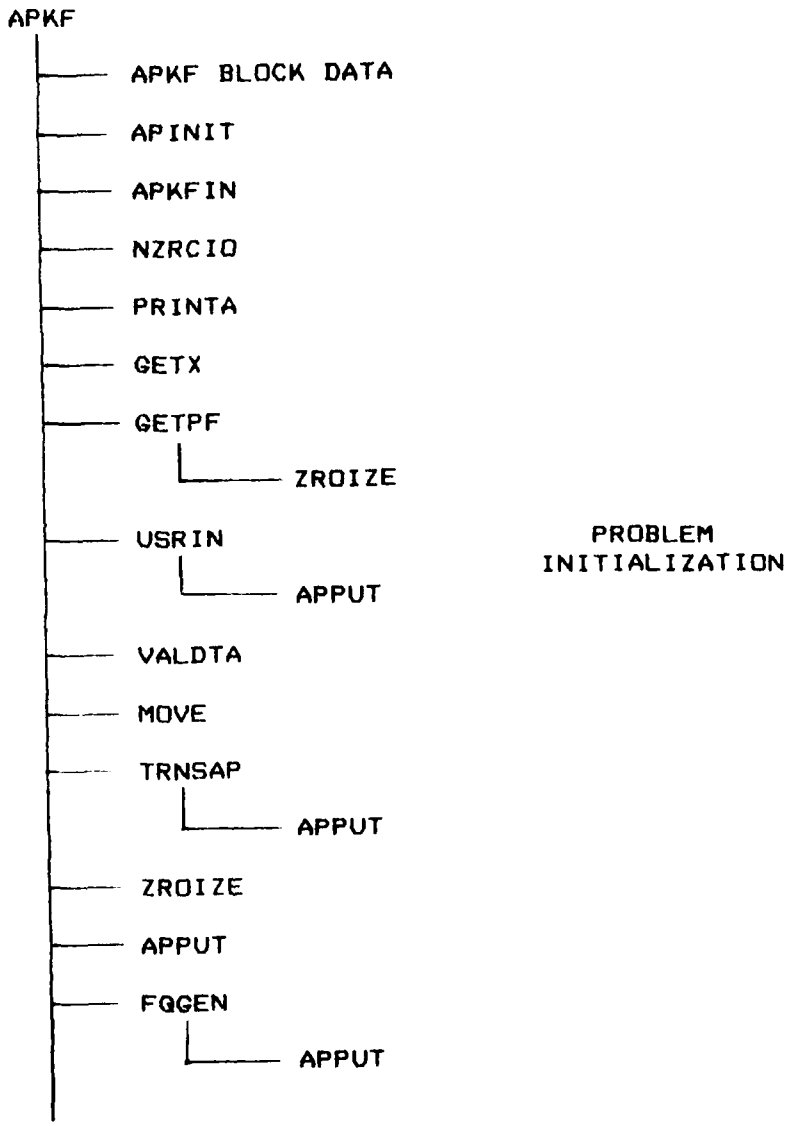
OUTPUT SWITCHES:

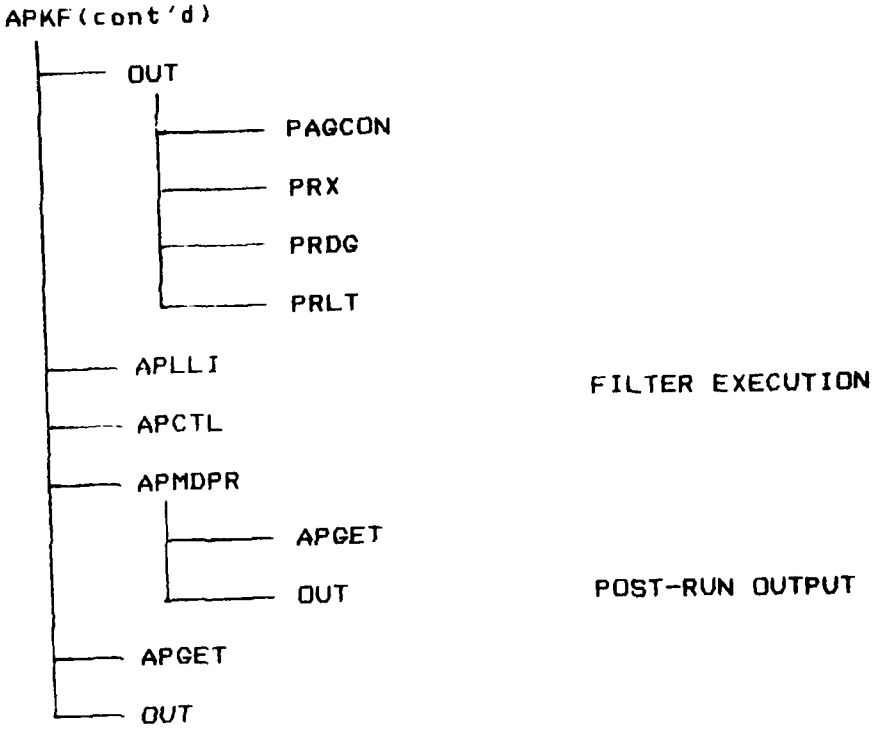
LPR	General Output
LPRXF	Filter State
LPRDG	Filter State Sigmas
LPRUD	General Update Output
LPRZR	Measurement Residual
LPRH	Measurement Observation Matrix
LPRK	Kalman Gain Matrix

AP COMMUNICATION:

IMSFIL	Pointer to Current Measurement
IFLPTR	AP MD Location of Measurement Pointer
IPRFIL	Pointer to AP MD Output File
IPRPTR	AP MD Location of Output File Pointer
IPRT	Update Output Switch
ILPRUD	AP MD Location of Output Switch
ISTAT	AP Hardware Status

2. APKF 11/70 FORTRAN SUBROUTINES




APKF(cont'd)
OUT
PAGCON
PRX
PRDG
PRLT
APLLI
APCTL
APMDPR
APGET
OUT
APGET
OUT
FILTER EXECUTION
POST-RUN OUTPUT

SUBROUTINE DESCRIPTIONS

11/70 ROUTINES

INPUT	PROCESS	OUTPUT

APKF

Sequences through filter initialization, transferring constants and initial filter values to the AP
Times the AP run and the total problem execution
Initiates execution of the AP filter
Calls OUT and APMDPR to print output listing

APCTL

Transfers the disk resident load module to the AP, putting overlay segments into MD
Loads the root segment into PS memory
Initiates execution of the AP code

APKFIN

Reads and prints date, time and values of run control variables
Checks vector dimensions

INPUT	PROCESS	OUTPUT
	APMDPR	
T	Transfers the filter output from a file in AP MD memory to the 11/70, and calls OUT to print it	
	FQGEN	
NZF NZQ	Calculates the non-zero elements of the matrices F and QF for propagation of the covariance matrix PF then transfers them to the AP	F QF
	GETPF	
NF NTR	Reads initial covariance matrix values	PF
	GETX	
NF	Reads initial state vector values	XF
	MOVE	
VECTOR-1 DIMENSION	Copies VECTOR-1 into VECTOR-2	VECTOR-2

INPUT	PROCESS	OUTPUT
	NZRCIO	
LABEL NZF, NZQ	Reads and prints the indices (row-column pairs) of the non-zero elements of a given sparse matrix	INDF, INDQ
	OUT	
TIME ID Global Switches and Variables	Based on ID and the output switch settings, writes filter values to a print file	
	PAGCON	
NLINES FLAG	If NLINES + current line count > 55 or FLAG =1, eject to a new page	
	PRDG	
T PF	Computes and prints the square roots of PF's diagonal elements	

INPUT	PROCESS	OUTPUT
	PRINTA	
NF M NZF NZQ NTR NALL	Calculates variable locations and prints the structure of the unlabeled common global area	
	PRLT	
T PF	Prints the symmetric matrix PF in a lower triangular display	
	PRX	
T VECTOR	Prints the values of a vector at time T	
	TRNSAP	
	Reads simulated sensor data from Logical Unit 8 and writes it to AP MD memory	

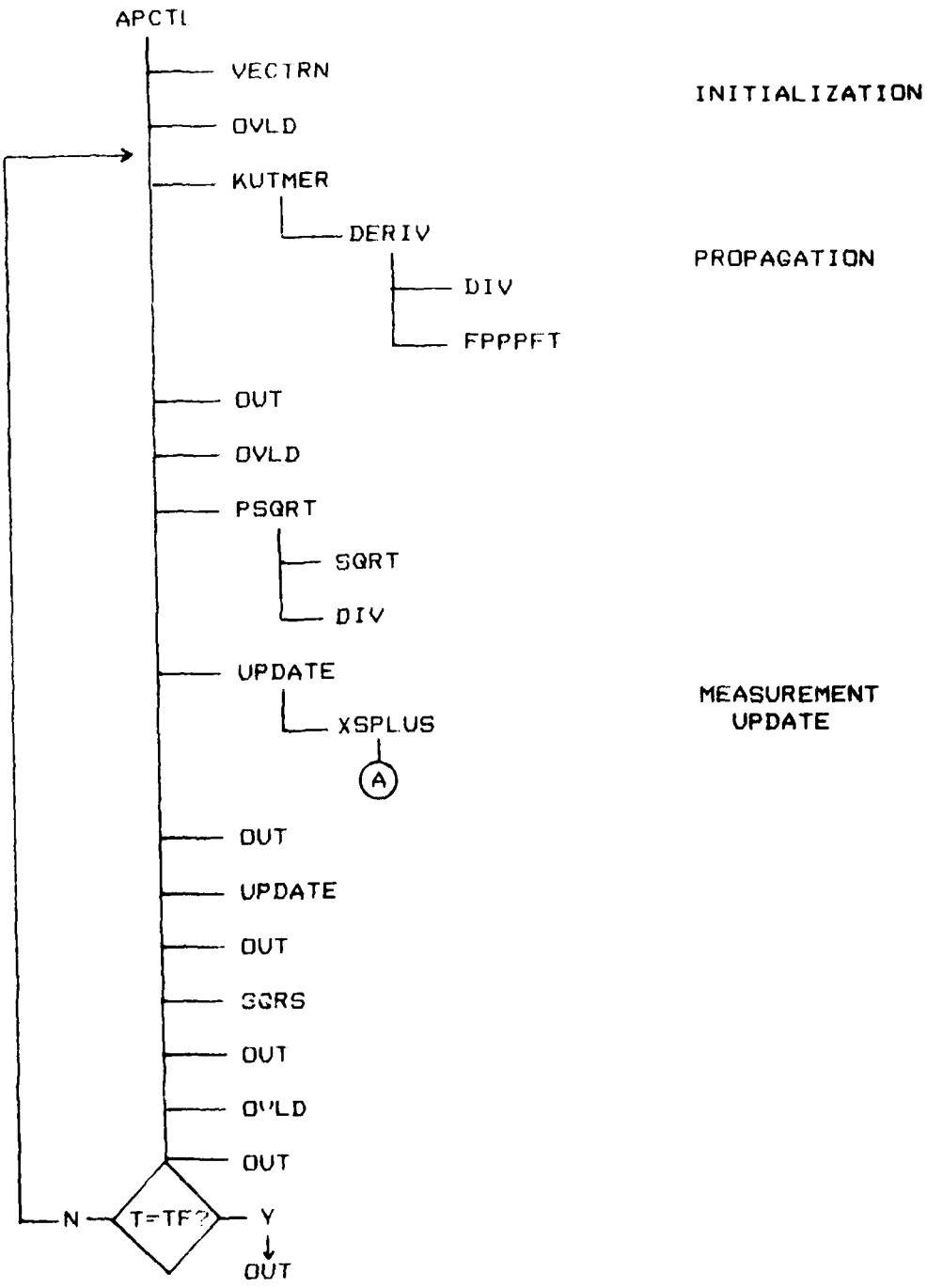
INPUT	PROCESS	OUTPUT
	USRIN Reads and prints supplied constants and transfers them to the AP	SDWF RFVCTR TAUF
TO TF TMEASO DTMEAS DTPRNT	VALDTA Checks and flags out-of-range parameters	
ROWS COLUMNS MATRIX	ZROIZE Zero's a given number of matrix elements	MATRIX

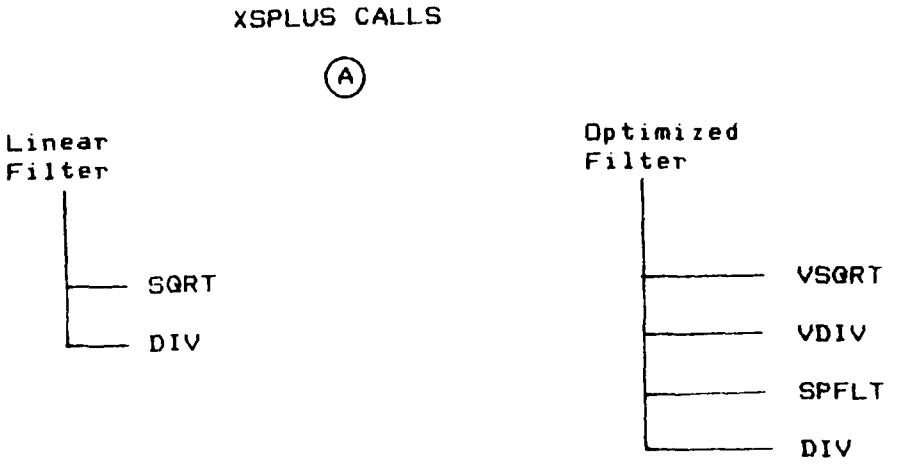
APEX UTILITY ROUTINES

INPUT	PROCESS	OUTPUT
	APGET	
VARIABLE NAME AP MD ADDRESS ELEMENT COUNT FORMAT SWITCH	Transfers the given number of AP MD words to the 11/70 locations indicated by VARIABLE NAME (FORMAT SWITCH: 1-Integer, 2-Floating Point)	11/70 VALUES
	APINIT	
ZERO ZERO	Waits until the AP is free, then reserves it for the calling program Hardware problems are indicated by ISTAT values Clears AP Main Data memory	ISTAT

INPUT	PROCESS	OUTPUT
	APLLI	
FILE NAME NAME LENGTH FILE LUN OPTION(=1) LOAD MODULE ID	Identifies the disk resident AP load module file name to HASI subroutine	
	APPUT	
VARIABLE NAME AP MD ADDRESS ELEMENT COUNT FORMAT SWITCH	APEX utility routine to transfer the given number of 11/70 words to AP Main Data memory	AP MD VALUES

3 APKF AP-120B APAL SUBROUTINES




XSPLUS CALLS
A
Linear Filter
SQRT
DIV
Optimized Filter
VSQRT
VDIV
SPFLT
DIV

AP-120B ROUTINES

INPUT	PROCESS	OUTPUT

APCTL

Transfers initial filter data from MD memory
  to Data Pad X
Loads filter propagation overlay
Calls KUTMER to propagate filter via numerical
  integration in steps of H0
At DTMEAS intervals:
  - Loads the update overlay
  - Updates the filter state vector and covariance
    matrix
  - Outputs update values based on MD switch LPRUD
  - Loads the propagation overlay
At DTPRNT intervals, outputs filter values
When T=TF, outputs final filter values and returns
  control to APKF in the 11/70

INPUT	PROCESS	OUTPUT
	DERIV	
XF PF QF G RE TAUF	Calculates the derivatives of the filter vector XF and the covariance matrix PF	XFDOT PFDOT
	FPPPFT	
F INDF PF	Computes and sums the first two terms in: PFDOT = F*P + P*FT + QF where FT = F-Transpose	PFDOT
	KUTMER	
XF(T) PF(T) T HO DEWK	Integrates a set of 1st-order ordinary differential equations from T to T+HO using a 5th-order Kutta-Merson numerical integration	XF(T+HO) PF(T+HO)

INPUT	PROCESS	OUTPUT
	OUT	
FILPTR IMEAS ID T H G ALPHA ZRES XF PF	Writes filter values to AP MD memory starting at the location designated by FILPTR The specific values written are determined by ID Updates FILPTR	MD FILE
	PSQRT	
PF	Calculates the Cholesky upper triangular square root matrix S of the covariance matrix PF	S
	SQRS	
S	Squares the uper triangular square root matrix S to reform the covariance matrix PF PF = S*ST where ST = S-Transpose	PF

INPUT	PROCESS	OUTPUT
	UPDATE	
UPSW APMS XF- S-	Reads a simulated sensor measurement at an MD location pointed to by APMS and calculates the measurement residual, ZRES Calls XSPLUS to update the filter state vector and the covariance matrix - UPSW indicates whether this update is for the 1st or 2nd measurement	ZRES XF+ S+
	VECTRN	
SWITCH	Based on the SWITCH setting, transfers XF and PF between MD and DPX	
	XSPLUS	
XF- S- H A G ZRES RFVCTR	Performs the sequential measurement updates on XF- and S- using the Carlson square root update equations	XF+ S+ ALPHA

APAL UTILITY ROUTINES

INPUT	PROCESS	OUTPUT
	DIV	
SCALAR-1 SCALAR-2	Performs a scalar division	QUOTIENT
	OVLD	
SEGMENT ID	Transfers the indicated overlay segment from MD memory to Program Source memory	
	SPFLT	
INTEGER	Transforms an SPAD integer into a floating-point number in DPX	FL-PT

INPUT	PROCESS	OUTPUT
	SQRT	
SCALAR	Calculates a scalar square root	SQRT(SCALAR)
	VDIV	
VECTOR-1 VECTOR-2	Calculates the quotients of the respective elements of the two input vectors	QUOTIENT-VECTOR
	VSQRT	
VECTOR	Calculates the square roots of the VECTOR elements	SQRT(VECTOR)

4. PROGRAM DEVELOPMENT AND VERIFICATION

The equations that make up the Kalman filter used for this project were developed as a sample problem to exercise a Kalman filter simulation environment called SOFE. Both SOFE and the sample problem were designed by Mr. Stan Musick of the Air Force Avionics Laboratory. Most of the FORTRAN routines used for APKF are to a large degree simplifications of his code, and the algorithms implemented here in APAL derive directly from the FORTRAN routines of SOFE and the sample problem. Besides executing the Kalman algorithm on a user's state vector, SOFE also propagates and updates a larger vector that represents the "truth state", or the actual values that the filter is meant to estimate. The filter state vector itself is a simplified version of the truth state, scaled to run in real time on limited hardware resources.

The first step in developing APKF was to eliminate from SOFE unneeded capabilities, and rewrite generally-applicable simulation software for the specific filter used. The resulting simulation was verified in a number of tests against the original. Next, this scaled-down simulation was transitioned from a CYBER 750 to a PDP 11/70. In order to check the reliability of the move, a file of random numbers was transferred also, and

used to run test problems on the 11/70. Results were identical to the degree expected given the machines' differing precisions.

After the 11/70 simulation had been verified, it was further scaled down and restructured to eliminate truth state propagation and derive measurement updates from a file of contrived sensor data written by the simulation. These simulated measurements were calculated by adding two elements of the truth state vector, one which represented the actual value of the quantity being measured, and another equal to the noise in the measuring device. This last version is a legitimate Kalman filter except for its lack of real time control. Instead, outputs and updates are scheduled based on parameters supplied at the start of each run. The results, in a variety of test scenarios for this FORTRAN filter, were identical to the results given by the simulation.

Finally, the actual filter algorithm was implemented in assembly language on the AP-120B. Only initialization and the printout of results remain in 11/70 FORTRAN routines. Results were compared between this dual-processor filter program and the all-FORTRAN version, with agreement found to six significant digits. Filter values were not identical for these last two versions because of the AP-120B's larger word size. As a last check, results for

the optimized APAL version were verified against those produced by the linear code. Only trivial differences appeared, these due to algorithm changes made to take advantage of the AP architecture.

A final interesting note is the comparison in program length between the FORTRAN Kalman filter implementation and both AP programs. Table 2 lists lines of code for all three versions organized by AP subroutine.

	11/70	AP-120B Assembly Language	
	FORTRAN	LINEAR	OPTIMIZED
KUTMER	51	247	249
DERIV	36	82	62
FPPPFT	37	105	102
PSQRT	33	93	93
XSPLUS	52	173	265
SQRS	21	63	62
TOTAL	230	763	833

Table 2. Lines of Code per Subroutine for all Three Versions

The substantial increase in size of XSPLUS derives from the use of Scheiss' algorithm restructuring to increase execution speed. A single loop with a recursive calculation that inhibited optimization was expanded to four loops, three of which became pipelined vector operations. The four loops increased routine length, but substantially reduced execution time.

SECTION VI

TIMING EXPRESSIONS

By executing individual subroutines on the AP-120B off-line simulator, or on the hardware using the debug interface from the 11/70, exact timing results can be obtained directly. However, one of the goals of this project was to provide a means of calculating execution times for filters of larger dimension than the one that was actually implemented. This requires the development of closed form expressions for execution time in terms of filter dimension.

Much of this task was straightforward. The AP-120B is a synchronous machine, so timing can be determined by simply counting instructions. Since the AP is a 6 MHz machine, dividing the instruction count by six gives execution time in microseconds. Simple loop times can be calculated as the product of iteration count, usually the filter dimension, and the number of loop instructions. With optimized loops, the setup and cleanup overhead becomes a constant term, and the loop size is multiplied by the filter dimension less a term that represents the iterations performed during setup and cleanup. Examples of this were given in Section 4.

AD-A119 221 AIR FORCE WRIGHT AERONAUTICAL LABS WRIGHT-PATTERSON AFB OH F/G 12/1
SOFTWARE OPTIMIZATION FOR ARRAY PROCESSORS. AN AP-120B KALMAN F--ETC(U)
JUN 82 E C DUDZINSKI

UNCLASSIFIED AFWAL-TR-82-1100 NL

2 OF 2
AD A
119221

END
DATE
FILMED
10-82
DTIC

In the Kutta-Merson calculations of the numerical integration, the vector operated on was a concatenation of the filter state vector and the upper triangular representation of the covariance matrix. This combined vector has a length of

$$(n^2 + n)/2 + n$$

so this is the iteration count that is multiplied by the loop length to calculate execution time.

Although the majority of expression determinations were like those above, exceptions were not uncommon. Three loop timing expressions that are characteristic of those exceptions are examined in the rest of this section.

1. INNER LOOP COMPLICATIONS

The first example is taken from subroutine XSPLUS, where the measurement update is done. A FORTRAN version of the loop under consideration is shown in Figure 20. The inner loop is executed "I" times for each of the N iterations of the outer loop, where I is the changing outer loop index.

```
      L = 0
      DO 4  I = 1,N
          A(I) = 0
          DO 4  J = 1,I
              L = L + 1
              SS = S(L)
              S(L) = AP(I)*SS - C(I)*A(J)
              A(J) = A(J) + SS*STHT(I)
4     CONTINUE
```

Figure 20. Covariance Square Root Update.

By charting the number of iterations of the inner loop for successive values of n, the filter dimension, a pattern is deduced and the expression n(n+1)/2 is derived to express it. Table 3 shows these results.

N	TOTAL INNER LOOP ITERATIONS	n(n + 1)/2
1	1	1
2	3	3
3	6	6
4	10	10
5	15	15
6	21	21
7	28	28

Table 3. Inner Loop Iterations

For the unoptimized APAL code, this expression made it easy to calculate loop timing as a function of filter dimension:

$$T = (i_o * n + i_i * (n(n+1)/2)) * 1\mu s/6$$

where i_o = the number of APAL instructions in the outer

loop, and i_j = the number in the inner loop. The problem came with optimization. Complications arose due to early exits from the cleanup code when the inner loop was iterated less than three times. As explained in Section 4, the setup and cleanup needed to fill the pipeline of an optimized loop will cause the calculation of a number of results in a single iteration of the loop. This particular code performs three calculations when the setup, loop and cleanup are executed once. It becomes necessary to jump out of the cleanup when the iteration count is one or two to avoid storing unintended and erroneous results. The setup for the optimized inner loop requires ten instructions, the loop itself is 5 instructions long, and cleanup is eight. When the iteration count of the inner loop is one, only two cleanup instructions are executed. When the iteration count is two, six cleanup instructions are necessary. When iterating three or more times, all eight are executed.

To derive a formula applicable to other filter dimensions, instruction counts were charted in Table 4, and the following expression derived:

$$T = 38 + 23*(n-2) + 5*[(n-2)(n-3)/2]*1\mu s/6$$

Listed below is a general form of this expression, applicable to all inner loops of the type considered here:

$$x + y * (n-i+1) + z*[(n-i+1)(n-i)/2]$$

where:

y = total number of instructions in setup, loop and cleanup,

x = cumulative inner loop instruction count prior to the first inner loop execution with no jumps out of the cleanup,

z = length of optimized loop, and

i = inner loop index the first time that the entire cleanup is executed

I	CLEANUP INSTRUCTIONS	INNER LOOP INSTRUCTIONS EXECUTED	INCREMENT	CUMULATIVE INNER LOOP INSTRUCTIONS EXECUTED
1	2	17	-	17
2	6	21	4	38
3	8	23	2	61
4	8	28	5	89
5	8	33	5	122
6	8	38	5	160

Table 4. Optimized Inner Loop Instructions

2. INTERLEAVE VIOLATIONS

Another unorthodox timing calculation arises from interleave violations, which occur commonly in loops as short as two or three instructions. In the STHT calcula-

tion of XSPLUS, shown in Figure 21, elements of the vectors H and S are read in consecutive instructions of the optimized loop. The five elements of H are accessed in order starting over with H(1) on each loop iteration, but the fifteen elements of S are read consecutively from the first to the fifteenth regardless of loop iterations. This makes it impossible to locate either vector in MD memory to completely eliminate interleave violations. Without belaboring the point, Table 5 shows the number of interleave violations that occur for a given filter dimension with both possible memory placement situations. The column following the interleave violation counts, which were calculated by hand, is the value of the expression that was inductively determined to express the violations as a function of filter dimension. Given both memory location situations, this is the best possible single estimator since it averages the number of violations that occur in each situation. It provides exact results for the sample filter, since a different H vector is used for each of the two measurement updates, and one starts in the same bank as S and the other does not.

```
      K = 0
      DO 20 I=1,N
            STHT(I)=0
            DO 20 J=1,I
                  K=K + 1
                  STHT(I)= STHT(I) + S(K) * H(J)
20    CONTINUE
```

Figure 21. STHT Calculation

FILTER DIMENSION(n)	INTERLEAVE VIOLATIONS WITH H(1) AND S(1) IN: SAME MEMORY BANKS	OPPOSITE MEMORY BANKS	ESTIMATOR EXPRESSION $n^2/4 + n/4$
1	1	0	.5
2	1	2	1.5
3	1	5	3.0
4	5	5	5.0
5	10	5	7.5
10	27	28	27.5
15	52	68	60.0
20	105	105	105.0
50	637	638	637.5

Table 5. Interleave Violations

3. SUMMATION EXPRESSIONS

A final example is taken from the SQRS subroutine, which executes certain parts of its algorithm with a repetition count that is easy to express as a summation, but

more difficult as a function of n. Specifically, the formation of PF from S requires a calculation that is executed five times for the first element, four times for both elements two and three, three times apiece for the next three elements, and so on. For a five element state vector that has a fifteen-element covariance representation, the total number of iterations for the calculation is given as:

$$1*5 + 2*4 + 3*3 + 4*2 + 5*1 = 35$$

or in general form:

$$\sum_{i=1}^{n} (i * (n + 1 - i)).$$

To put this into a closed form expression, instruction counts for different values of filter dimension were calculated and expressed as functions of the dimension. The first two columns of Table 6 show this. The coefficient of n was quickly seen to be equal to n(n + 1)/2, a familiar form from inner loop iteration counts. It took longer to realize that the second term was nearly a third of the filter dimension cubed. From there, a little trial and error produced $(n^3 - n)/3$, which was exactly equal to the constant term. This resulted in the timing expression:

$$n * [n(n + 1)/2] - (n^3 - n)/3$$

which simplifies to:

$$(n^3 + 3n^2 + 2n)/6.$$

FILTER DIMENSION(n)	SUMMATION ITERATIONS	n^3
1	1 = n	1
2	4 = 3n - 2	8
3	10 = 6n - 8	27
4	20 = 10n - 20	64
5	35 = 15n - 40	125
6	56 = 21n - 70	216
7	84 = 28n - 112	343
8	120 = 36n - 168	512

Table 6. Summation As a Function of Filter Dimension

4. SOFTWARE GENERALITY

One last timing issue deals with the calculation of ALPHA in XSPLUS. The two-instruction optimized loop has an extensive setup and cleanup, so much so that a single trip through the loop completes calculations on six vector elements. Since our filter has a dimension of only five, some instructions were inserted as non-executable comments to document the optimized loop without actually performing

operations on the non-existent sixth element. NOP's were inserted so that the cleanup had the necessary length for vectors with a dimension greater than five. This resulted in two more instructions than required for the sample filter, but provided timing generality allowing extrapolation to any filter dimension.

One problem does arise, however. With the sample filter of dimension five, the loop was executed only once and the timing expression was a constant equal to the total instructions in setup, loop and cleanup. For larger filters, it is necessary to include the term 2*(n - 6) added to the constant factor to represent loop iterations. Hence, there are two closed form expressions for XSPLUS execution time, one for filters of dimension five or less, and one for those that are larger.

Timing expressions for each subroutine will be presented in the next section, along with the speedup achieved for specific test scenarios.

SECTION VII

OPTIMIZATION RESULTS

Software optimization for the AP-120B was done for the complete 5-state Carlson square root filter described in Section 5. First a linear version of the filter was written, programming the AP without taking advantage of its parallel functional units, pipelined adder and multiplier, or independent data flow circuitry. This version is intended to simulate execution on a conventional machine architecture with the same physical logic gate speed as the array processor.

Optimization was done on this sequential program using all of the techniques described in previous sections. The update algorithm was restructured in accordance with the example of Section 4, resulting in nine independent loops suited for pipelined calculation. Cache register usage was maximized to the extent that the filter state vector and the covariance matrix were stored permanently in the CPU data pads. Arithmetic expressions were rearranged to maximize parallelism, and independent operations were generally overlapped as much as possible.

Most importantly, all computational loops were "wrapped" into minimum size. In only one case was it im-

possible to achieve the theoretical minimum instructions per loop, and that was due to register access problems.

Twice, loops were coded with more instructions than necessary. This occurred when loop setup initiated computations for all five of the array elements and no further iteration was needed. However, an extra instruction for iteration control was included even though the loop test was always negative. The purpose of the extra instruction was to allow extrapolation of execution time to filters of larger dimension.

The biggest problem of the optimization was the inner loops iterating from one to the outer loop index, which itself increased from one to the filter dimension. As described in Section 4, minimizing these loops required an extra counter to exit the cleanup before storing unintended results.

The last three parts of this section will detail the actual execution speedup effected by the optimization. First, measured execution times will be given for each subroutine of the sample filter, along with extrapolated timings for a filter with the more characteristic dimension of twenty. Then timing expressions as a function of filter dimension for both linear and optimized versions of each subroutine will be presented.

Following the individual subroutine treatment, test

scenarios that were run on the sample filter will be described, and the results listed.

Finally, calculated execution times for the most rigorous test scenario will be presented for a variety of filter sizes. Calculations will be made for both linear and optimized filters. These figures will enable a determination of the actual speedup possible in an airborne filter without the overhead of our multi-processor, limited-memory AP system.

1. INDIVIDUAL SUBROUTINE RESULTS

Table 7 lists execution times by subroutine for both software versions and two filter dimensions. Also included is the ratio of the linear and optimized times, which records the speedup achieved.

The rest of this section will present the closed form expressions for each subroutine, along with a few notes of explanation. Numbers calculated from the expressions represent instruction counts, they must be divided by six to yield the actual execution time in μs. In all the formulas, "n" represents the filter dimension.

	EXECUTION TIME(s) DIMENSION=5			EXECUTION TIME(s) DIMENSION=20		
SUBROUTINE	LINEAR	OPTIMIZED	SPEEDUP	LINEAR	OPTIMIZED	SPEEDUP
APCTL	25.17	20.83	1.21	35.17	23.33	1.51
OUT	31.83	19.33	1.65	204.33	106.83	1.91
VECTRN	24.33	4.83	5.04	269.33	39.83	6.76
KUTMER	482.67	91.50	5.28	5417.67	791.50	6.84
DERIV	22.00	19.17	1.15	87.00	76.67	1.13
FPPPFT	204.67	17.00	12.04	10324.67	1028.25	10.04
PSQRT	215.83	194.00	1.11	6075.83	4339.00	1.40
UPDATE	2.83	2.83	1.00	7.83	7.83	1.00
XSPLUS	227.08	135.83	1.67	1858.33	669.25	2.78
SQRS	140.50	137.17	1.02	4591.33	4553.00	1.01

Table 7. Actual and Extrapolated Execution Times

APCTL

LINEAR: $(39 + 4n)*u + 12o + 40i + 40$

OPTIMIZED: $(39 + n)*u + 11o + 32i + 38$

where u = measurement updates

o = scheduled filter outputs

i = integration steps

Execution times of Table 7 reflected a single iteration of each control segment. Very little optimization was possible in APCTL because there is only one vector operation. What speedup was achieved resulted from reducing the loop which resets the filter state vector from four instructions to one. Any other savings came from simply overlapping a few independent floating point operations.

VECTRN

LINEAR: $3.5n^2 + 10.5n + 6$

OPTIMIZED: $0.5n^2 + 1.5n + 9$

VECTRN is simply a single loop to read the initial filter state vector and covariance matrix values from MD memory and store them in Data Pad X. It was easily optim-

ized.

OUT

ID	LINEAR	OPTIMIZED
2	$2n^2 + 3n + 20$	$n^2 + 2n + 19$
6, 10	$2n^2 + 3n + 29$	$n^2 + 2n + 28$
7	$12n + 40$	$6n + 36$
11	$4n + 29$	$2n + 28$
12	$2n^2 + 3n + 23$	$n^2 + 2n + 22$
LPRUD=0	18	17

The ID value passed to OUT determines which of three sets of filter values will be written to MD memory. If LPRUD = 0, OUT will write nothing when called during the measurement update, returning after executing 18 instructions. Table 7 execution times represent the output of all three sets of data.

KUTMER

LINEAR: $70.5n^2 + 211.5n + 76$

OPTIMIZED: $10n^2 + 30n + 149$

The numerical integration was very amenable to optimization. It is essentially five loops, and each loop operates on the concatenation of XF and PF. Consequently, the long vector operations yield big time savings once optimized.

DERIV

LINEAR: $26n + 2$

OPTIMIZED: $23n$

Calculation of the filter state vector derivatives requires a different equation for each element. Hence, there are no vector loops to optimize.

FPPPFT

LINEAR: $7n^3 + 15n^2 - 2n - 12$

OPTIMIZED: $0.75n^3 + 0.5n^2 - 1.75n + 4.5$

FPPPFT determines part of the derivative of the symmetric covariance matrix which is stored as an upper triangular vector representation of length $(n^2 + n)/2$. This storage scheme reduces memory usage and eliminates duplicate computations. It further speeds up both linear and

optimized code by allowing permanent storage of this short vector representation in the CPU Data Pads.

The dramatic speedup in FPPPFT was caused by algorithm changes. In the original FORTRAN filter and in the linear APAL code, a general purpose sparse matrix algorithm was used It was primarily made up of integer operations to calculate indices into the upper triangular covariance vector PF and the sparse transition matrix F. Since integer operations cannot be pipelined, the routine could be optimized only slightly in its original form. In the first optimized filter, this routine was left largely alone.

However, when tests were run on this first optimized version, the speedup was only 1.5 for a five-element vector, and became less as vector dimension increased in the extrapolations. Analysis showed that 80% of execution time for the five-state filter was being spent in FPPPFT, with the percentage increasing to 97% for a twenty-element vector. Since the size of the covariance matrix and the number of operations in FPPPFT grows with the square of the vector dimension, this relatively unoptimized general purpose routine neutralized the speedup in other parts of the filter.

The solution chosen was to code the specific element multiplications without any algorithm generality. The im-

provement was marked.

PSQRT

LINEAR: $2.5n^3 + 41.5n^2 - 6n - 25$

OPTIMIZED: $n^3 + 45.5n^2 - 4.5n - 76$

Determining the Cholesky decomposition of P, where P is stored in upper triangular mode, requires predominantly integer calculations. There is only a single loop to optimize.

UPDATE

LINEAR AND OPTIMIZED: $7 + 5(2n/5)$

UPDATE contains no loops whatsoever. Operations are dependent and permit no overlap.

XSPLUS

LINEAR: $19.25n^2 + 171.25n + 25$

OPTIMIZED: $4.75n^2 + 91.75n + 237.5$ for $n \leq 5$

$4.75n^2 + 94.75n + 220.5$ for $n > 5$

This subroutine was by far the hardest to optimize. It included inner loops iterating to an outer loop index and optimized loops that performed more calculations on a single iteration than required for the 5-state sample filter. The latter complication forced the second timing expression for optimized code, as explained in Section 6.

Even though the speedup achieved for XSPLUS was the most difficult, it was also very effective due to the large number of vector operations in the measurement update, especially after applying the algorithm restructuring.

SQRS

LINEAR: $2.67n^3 + 14n^2 + 30.33n + 8$

OPTIMIZED: $2.67n^3 + 13.5n^2 + 28.83n + 8$

Just as with PSQRT, this squaring of the covariance matrix stored as an upper triangular vector requires a preponderance of integer operations which resist optimization.

2 TEST SCENARIOS

Six different tests were run and timed. Since there is no accessible clock in the AP-120B, timing calls were made from the 11/70. This was unfortunate because it forced the inclusion of all AP overhead in the measured execution times. A single call from the FORTRAN HASI routine writes the APAL load modules to AP MD memory and transfers control to the AP. Here a system utility routine loads the root overlay segment into PS memory and begins filter execution. All of this transfer overhead as well as the swapping of overlay segments done by APCTL during the run itself is included in the execution times measured from the 11/70 FORTRAN program.

Because of this overhead inclusion, the test run results do not show the optimization improvement to the degree calculated by the timing expressions. In fact, since the simulator can provide all of the timing results that are necessary, the primary purpose of the test runs was to debug and demonstrate the complete two-processor filter task. Individual subroutine timing results would be meaningless if the program that these subroutines comprise were not a correct one.

The test scenarios differed in duration, event scheduling and filter output. The different event intervals

resulted in varying patterns of subroutine execution and overlay swaps, and of course in different filter values. These event intervals determined how long filter values were propagated via numerical integration before stopping for output or a measurement update.

There were two options for filter output during AP runs. Test runs with the "update" option wrote the filter state vector and the covariance matrix values to MD before and after each set of measurement updates. They also output the Kalman gain matrix, the measurement observation matrix, the measurement residuals, and the standard deviation of those residuals during the updates. Finally, they wrote the filter state values following the feedback reset. All of these values are eventually printed by the host FORTRAN program. Runs with the "no update" output option call OUT during the update, but a switch set in APCTL at run initialization prohibits writes to MD and forces an immediate return.

All test runs write state vector and covariance matrix values to MD at the scheduled output intervals, and they all eventually print out a listing from the FORTRAN code showing the state vector values at the scheduled output times and the state vector error sigmas, which are the square roots of the diagonal elements of the covariance matrix. Listings for runs with the "update" output option

include a printout of the entire covariance matrix at each scheduled output.

Table 8 details the scenario differences by listing output option and event intervals.

TEST NR	RUN DURATION (seconds)	OUTPUT INTERVAL (seconds)	UPDATE INTERVAL (seconds)	UPDATE OUTPUT (Y=YES, N=NO)
1	108,000	3,600	3,600	Y
2	36,000	1,200	1,200	Y
3	108,000	10,800	90	N
4	36,000	3,600	30	N
5	3,570	1,200	1,200	Y
6	1,080,000	36,000	1,080	N

Table 8. Test Scenarios

The greater the frequency of updates in the scenarios shown, the greater the overlay handling overhead in the results. APCTL and OUT are always kept in PS memory, and the numerical integration routines are placed in PS memory at the start of the run and left there until an update event. At that time, they are swapped out to make room for the update routines. After the update, the numerical integration code is swapped back in.

In order to determine the amount of overhead in each of the test runs and also to extrapolate a given test case to filters of larger dimension, a small program was designed and written to calculate AP run times. A control scheme identical to that of APCTL reads the test parame-

ters and steps through each scenario, accumulating execution time using the closed form expressions of this section. Two versions of this program compute both linear and parallel timing results.

Table 9 shows test results for all six scenarios and three filter versions. It also lists the calculated execution times that don't include the load module transfer and overlay handling operations. For the very short test five, overhead involved in transferring the larger load module of the optimized program from the 11/70 to the AP overcame the time savings achieved by that code. In all but test six, the overhead is so large a percentage of execution time that it obscures the speedup achieved.

To show the optimization results for extrapolated filter sizes, test scenario six was run in the timing program for five different filter dimensions. Those results appear in Table 10. The execution time ratio in column four documents the speedup that resulted from the software optimization efforts of this project.

	EXECUTION TIME (seconds)				
	PDP 11/70	LINEAR APAL		OPTIMIZED APAL	
TEST	FORTRAN	MEASURED	CALCULATED	MEASURED	CALCULATED
1	134.401	7.219	5.869	2.684	1.020
2	47.079	3.281	1.975	1.969	0.354
3	158.201	9.035	6.851	4.567	1.756
4	69.273	5.084	2.957	3.833	1.089
5	4.040	1.480	0.195	1.603	0.034
6	1175.205	61.950	59.258	13.965	10.626

Table 9. Test Run Results

FILTER DIMENSION	CALCULATED EXECUTION TIME (SECONDS) LINEAR APAL	OPTIMIZED APAL	SPEEDUP FACTOR
5	59.258	10.626	5.58
10	320.669	41.599	7.71
20	2083.815	237.844	8.76
50	28682.333	3148.949	9.11
100	219953.461	24062.035	9.14

Table 10 Extrapolated Filter Execution Times

SECTION VIII

CONCLUSIONS

1 PROGRAMMING THE AP-120B

As with any program development, the algorithm should be completely defined, program structure determined, and memory usage mapped prior to writing a line of assembly language code. An essential part of the algorithm definition is its detailed expression in a high-order language or pseudo-code, from which the assembly language routines can be conveniently coded. The development time of Table 11 encompasses programming time after the algorithm had been so expressed. In fact, a complete working FORTRAN program provided the blueprint for APAL development.

	MANHOURS	LINES OF CODE PER MANHOUR
CODE	120	8.7
TEST	95	11.0
DEBUG	40	26.0
OPTIMIZE	40	26.0
TOTAL	295	3.5

Table 11. Manhour Expenditure for Software Development (An additional 40 hours were spent getting undocumented system software working)

Having available FORTRAN code proved invaluable in

debugging and testing the AP routines By printing out intermediate values of the variables in a run of the FORTRAN program, it was an easy task to check each subroutine in the simulator on the same test data After that process ferreted out any coding errors, the entire filter was run with one APAL routine at a time replacing the analagous FORTRAN code Thus ten versions of the complete filter were implemented, each increasingly composed of APAL code Optimization of the final filter was done a routine at a time, with the complete set of tests run on each optimization version The best way to remove errors from a finished program is to make sure they never get in.

2 PROGRAMMING THE ARCHITECTURE

Writing AP code is not like programming any conventional machine Complete knowledge of the architecture is required In fact, a detailed diagram of functional units and bus structure to determine allowable operations is a prerequisite for coding. The other essential aid is a breakdown of op-code fields to determine which operations can be specified simultaneously and which have overlapping fields Accepting the necessity to program the architecture as well as the algorithm, the programmer should have

few problems with the instruction set. My preparation for writing a thousand lines of complex AP code was a five-day course at the manufacturer's facility, and I found this perfectly adequate. In fact, the only real debugging necessary involved the coordination of Data Pad index values, register usage and pipeline pushers in tightly-wrapped optimized loops. With trivial exceptions, code outside of loop optimizations ran correctly as first w.itten. Minimization of loop length, however, was definitely non-trivial.

3. SYSTEM PROBLEMS

I wish there had been as few problems with the AP system software. Overlay use was virtually undocumented. As described in Section 2, implementing the overlay structure of this filter was largely a trial-and-error task. Loader input to generate an overlay structure and register use by the overlay operation were serendipitously determined. And once the overlay tables and segmentation had been properly created, there was no capability to exercise that overlaid code on the simulator or via the hardware debug interface.

Other system operations that caused problems were

also due to lack of documentation. System routines that loaded AP object modules made use of Data Pad registers without saving their current value, requiring that the Data Pad index be set to a scratch area if Data Pad values had to be saved during load module changes.

Also, I was using a new version of the AP software in order to use overlays, and this version required different logical unit numbers for its own use than the previous version. The AP diagnostics when I tried to use these LUN's in my filter program were not only uninformative, but downright deceptive. Only examination of the FORTRAN source for the new AP system software uncovered the problem.

The system problems just mentioned constituted the only extended delays in the entire software development.

4. AP-120B ARCHITECTURE

There are only a few changes I'd like to see in the architecture of the AP-120B and its instruction set. The memory address register requires the DP bus to load an immediate value, but this same bus transports all Data Pad register values to the memory input register. This means that two instruction cycles are necessary to store a Data

Pad value into MD memory if the destination address must be loaded from the immediate field. A separate immediate value-memory address bus would allow single-cycle operation. Since array operations usually fetch or store using incremented address values in an SPAD register or the memory address register itself, this is generally not a problem inside of loops.

Separate primary Data Pad indices for DPX and DPY would be very helpful. Since I stored the state vector and covariance matrix permanently in DPX and DPY, it was difficult to access values stored in one area of the Data Pads while operating on filter values kept in a different area. It was often necessary to change the DP index, add zero to the required DP value in the floating adder, change the DP index again, then push out the floating adder result in order to fetch an operand needed for filter calculations.

In an architecture designed for an airborne filter, I expect this problem would be eliminated by having cache register storage for filter values entirely separate from general purpose registers.

Lastly, if immediate values could be directly added to or subtracted from SPAD integer registers it would eliminate the involved procedure of first loading the immediate value into an integer register then performing a

register-register operation

In sum, I liked both the AP architecture and its instruction set. While responsibility falls on the programmer's shoulders to coordinate the independent operations, exceptional flexibility and processing power is made available at the same time. Remarkable speedups can be achieved for floating point operations on long vectors.

Although correctly coding an optimized loop for the AP-120B is much more difficult than writing a loop for the standard AF 1750 architecture, the AP is a synchronous processor with well-defined machine states that allows program testing every bit as rigorous as that possible on the 1750. Because of this, software written for it can be verified just as reliably as the code being flown today, although development time will be longer.

For adequate assessment of the AP-120B, I'd like to program this filter on some alternative array architectures. Machines with vector functional units like the Cray-1 perform a vector operation with a single instruction, rather than requiring an optimized loop. However, while a loop can accommodate any vector size, a vector functional unit operates on only a predefined number of elements, so very long vectors require repeated operations. The overhead could become onerous, and the Cray-1 sacrifices flexibility for its programming ease.

Multiple processors represent another approach to array processing, in two different configurations. First, the SIMD machine uses different ALU's to perform identical operations on multiple vector elements. Secondly, a Kalman filter can be designed in discrete segments that execute in parallel on different processors, with the results synthesized in a third operation. These approaches could be combined using a number of coordinated processors in what might be the best way to maximize filter speed. The problem is one of data flow and reliability. I feel much more comfortable using a pipelined architecture or vector functional units than I would trying to synchronize multiple processors.

5. THE BOTTOM LINE

Compilers to generate optimized horizontal microcode are essential. Writing software in assembly languge for an array processor is expensive, in terms of both time and personnel. While this might be feasible for isolated military applications, widespread commercial use of array processor power will hinge on development of a reliable compiler for a High Order Language.

Cray and Floating Point Systems both have FORTRAN

compilers, but much work needs to be done before they can approach the efficiency of handwritten assembly language. While these compilers, coupled with vast libraries of optimized subroutines, are sufficient for many applications, speed requirements in the future will be increasingly rigorous.

The capability for developing an optimizing compiler certainly exists. If the technique for wrapping AP loops can be described and taught, it can also be programmed. Restructuring an algorithm to make it more amenable to vectorization will be the responsibility of the system analyst, but the tedious task of row/column wrapping, register allocation and generating setup and cleanup code is best left to efficient system software .

6. AIRBORNE AP'S

The degree of speedup demonstrated here for array processor use in a Kalman filter application justifies the premise of this project: airborne AP's can markedly enhance processing power in the cockpit. With the very large reductions in execution time that resulted from this project's optimization, embedded array processors could allow next-generation Kalman filters to fuse the multitude

of available navigation data and still have power unused to enhance other avionics operations.

From the point of view of software complexity and program reliability, pipelined vector architectures or vector functional units would be preferable to a SIMD multi-processor configuration. And of those two alternatives, a pipelined machine would require less hardware.

In fact, an architecture very much like the AP-120B's would appear to be one of the better choices as a model for an embedded avionics processor. Its independent functional units, synchronous design and pipelined vector operations provide the necessary processing speed without sacrificing flexibility or reliability, and although programming it optimally is not simple, it is certainly within the scope of Air Force avionics software development.

REFERENCES

Alspach, Daniel L. A Parallel Processing Solution to the Adaptive Kalman Filtering Problem with Vector Measurements, AFOSR-TR-73-2238, Fort Collins, Colorado: Colorado State University, 1973.

Calahan, D. A. Algorithmic and Architectural Issues Related to Vector Processors, Report No. AFOSR-TR-76-0870, University of Michigan, Ann Arbor, Michigan, 1976.

Carlson, Neal A. "Fast Triangular Formulation of the Square Root Filter," AIAA Journal, Vol. 11, No 9(September 1972), 1259-1265.

Du Plessis, Roger M. Poor Man's Explanation of Kalman Filtering or How I Stopped Worrying and Learned to Love Matrix Inversion, Anaheim, California: Autonetics Division, North American Rockwell Corporation, 1967.

FPS Technical Publications Staff. Processor Handbook, No. 60-7259-003C, Beaverton, Oregon: Floating Point Systems, Inc., 1980.

FPS Technical Publications Staff. Programmers Reference Manual Part One, No. 800-7319-000, Beaverton, Oregon: Floating Point Systems, Inc., 1980.

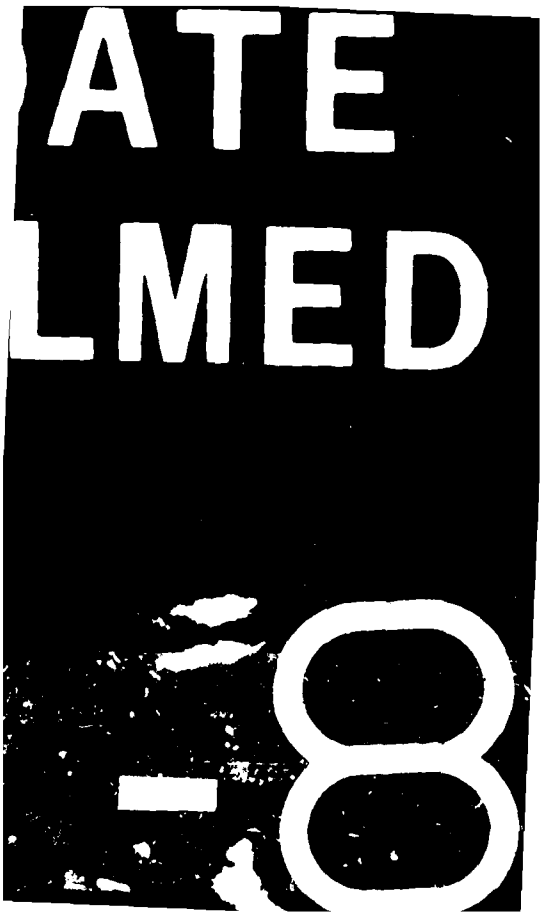
Kuck, David J. The Structure of Computers and Computations. Vol. 1, John Wiley and Sons, 1978.

Maybeck, Peter S. The Kalman Filter, An Introduction for Potential Users, AF Flight Dynamics Laboratory, AFSC, WPAFB, 1972.

Maybeck, Peter S. Stochastic Models, Estimation and Control. Vol. 1, Academic Press, 1979.

Musick, Stanton H. SOFE: A Generalized Digital Simulation for Optimal Filter Evaluation User's Manual. AFWAL-TR-80-1108, AF Avionics Laboratory, AFSC, WPAFB, 1980.

Scheiss, James R. Vectorization of Linear Discrete Filtering Algorithms. Report No. NASA TM X-3527, Washington, D.C.: National Aeronautics and Space Administration, 1977.


ATE
LMED
-8