Parameterizable FPGA-based Kalman Filter Coprocessor Using Piecewise Affine Modeling

Aaron Mills, Phillip H. Jones, Joseph Zambreno
Iowa State University, Ames, Iowa, USA
{ajmills,phjones,zambreno}@iastate.edu

***Abstract*—The Kalman Filter is a robust tool often employed as a plant observer in control systems. However, in the general case the high computational cost, especially for large system models or fast sample rates, makes it an impractical choice for typical low-power microcontrollers. Industry trends towards tighter integration and subsystem consolidation point to the use of powerful high-end SoCs, but this complicates the ability for a controls engineer to verify correct behavior of the system under all conditions, which is important in safety-critical systems.**

Dedicated FPGA hardware can provide computational speedup, in addition to firmer design partitioning in mixed-criticality systems and fully deterministic timing, which helps ensure a control system behaves as close as possible to offline simulations. We introduce and compare two variants of a software-configurable FPGA-based implementation of a Kalman Filter. The first is an implementation of an Extended Kalman Filter, while the second is a novel approach–the Piecewise-Affine Kalman Filter–which may offer significant advantages for certain types of applications.

The state estimate update time and resource requirements are analyzed for plant models up to 28 states. For large models, the designs provide a speedup of 7-12x compared to reference ARM9020T software implementations. An application-agnostic performance analysis demonstrates how the Piecewise-Affine Kalman Filter reduces the software workload and the communication overhead compared to the standard mixed hardware-software Extended Kalman Filter approach.

I. Introduction

The Kalman filter is a common means of accurately estimating the state of a plant in the presence of noise in either measurements or the model itself. As discussed in [1], the computational complexity of the Kalman Filter, when coupled with the high-dimensionality of models and high sample rates required in applications such as inertial navigation systems (INS), would tend to overwhelm processing resources available in the low power microcontrollers used in typical automotive or industrial settings. On the other hand, high-end SoCs are attractive for their ability to time-multiplex many different tasks, but this approach makes control validation more difficult due to unmodeled, emergent timing characteristics. For some applications, thermal constraints may also impose limits on overall power.

Field-programmable gate arrays (FPGAs) are of growing interest in the area of applied control theory [2]. In addition to the massive parallelism available on FPGAs that can be utilized to obtain high controller update rates, software-hardware co-design using FPGAs can help separate embedded software concerns (e.g. real-time scheduling feasibility), from controls concerns (e.g. accounting for update-rate jitter). Kozak in [3] suggested that a software-hardware co-design approach for implementing advanced controllers in FPGAs would enable designers to make better use of complex controllers in high-speed systems. Monmasson [4] makes a similar suggestion, pointing out how different parts of a control algorithm are better suited for different types of hardware. In [5] a call is made for designs that make efficient use of the parallelism available on FPGAs, while retaining the generality and flexibility available to software solutions. Our work pursues this goal by focusing on an application-agnostic architecture.

Mixed hardware-software implementations of the Extended Kalman Filter (EKF) have been proposed in existing literature [1], [6], [7], [8], [9], which focus on computational acceleration. These designs consist of an application-specific, nonlinear, non-acceleratable part that is computed in software, and a generic matrix-math part which is computable in hardware. We claim that the construction of existing approaches introduces an artificial upper bound on both the overall speedup, and fail to explore the broader architectural advantages for control systems and mixed-criticality systems. Here we primarily focus on the first claim.

As an alternative to focusing on the EKF algorithm, we propose a piecewise affine modeling approach which uses the standard linear Kalman Filter. This not only offers an application-layer speedup over the EKF approach, but allows the complete plant observer to be computed on the coprocessor in a general way, thereby dramatically simplifying the analysis of control loop timing. The high level structure is illustrated in Fig. 1, along with functional units proposed in [10].

Our contributions consist of: 1) an implementation of the FPGA architecture supporting the piecewise affine approach, 2) an assessment of the scalability of the implementation (as it has not been explicitly analyzed in prior work), and 3) an exploration of the performance tradeoffs between the typical mixed hardware-software EKF approach and the proposed piecewise affine approach.

The remainder of this paper is organized as follows. In Section II, we discuss related research in this problem space. In Section III, we provide design details of our software configurable coprocessor, and provide an overview of our architecture. In Section IV, we describe the FPGA resources required for implementation and characterize the performance of the piecewise-affine approach. Section V concludes this paper and provides avenues of future work.

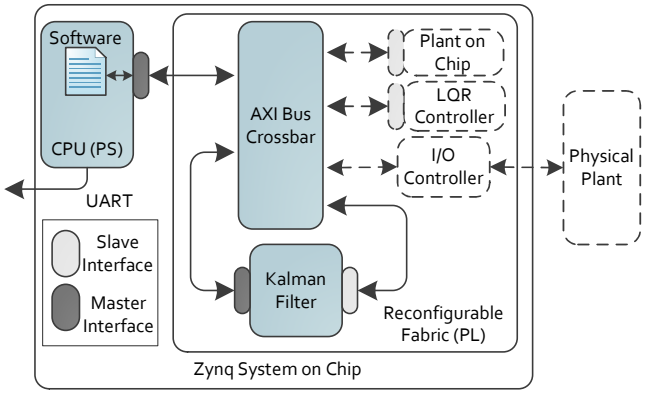

Fig. 1. System context and hardware-software interconnects.

II. Background

A. Kalman Filter and Extended Kalman Filter (EKF)

The Kalman filter, which only applies to linear models, involves the repeated application of the following steps at a particular sampling rate:

1) Estimate state
2) Calculate error covariance
3) Calculate Kalman gain
4) Update state estimate based on measurement
5) Update error covariance based on measurement

This process is shown more formally in lines 5 through 9 of Algorithm 1. In this paper, the subscript k indicates the discrete time-step in question. Furthermore the notation x_k^- denotes the state vector *before* performing the measurement update, and x_k^+ to denote the state vector *after* performing the measurement update.

Algorithm 1 Kalman Filter

1: **procedure** INITIALIZE
2: $\hat{x}_0^- \leftarrow E[x_0]$
3: $P_{\tilde{x},0}^- \leftarrow E[(x_0 - \hat{x}_0^-)(x_0 - \hat{x}_0^-)^T]$
4: **procedure** UPDATE
5: $\hat{x}_k^- \leftarrow f(\hat{x}_{k-1}^+, u_{k-1})$ # Estimate state
6: $P_k^- \leftarrow A_{k-1} P_{k-1}^+ A_{k-1}^T + P_w$ # Calc. error cov.
7: $K_k \leftarrow P_k^- C_k^T [C_k P_k^- + P_v]^{-1}$ # Calc. gain
8: $\hat{x}_k^+ \leftarrow \hat{x}_k^- + K_k[y_k - g(\hat{x}_k^-, u_k)]$ # Update state
9: $P_k^+ \leftarrow (I - K_k C_k) P_k^-$ # Update cov.

In the EKF, which extends the Kalman Filter to non-linear models, the derivative of the state update and measurement equations $f(\hat{x}_k, u_k)$ and $g(\hat{x}_k, u_k)$ (Equations 1 and 2) must be determined in advance analytically. During each update step, they are evaluated at the current state estimate.

$$A_{k-1}^x = \left.\frac{\partial f(x_{k-1}, u_{k-1})}{\partial x_{k-1}}\right|_{x_{k-1}=\hat{x}_{k-1}^+} \quad (1)$$

$$C_k^x = \left.\frac{\partial g(x_k, u_k)}{\partial x_k}\right|_{x_k=\hat{x}_k^-} \quad (2)$$

Overall these expressions allow us to produce a first-order (linear) estimate of function behavior at the current state value, but further increase the computational effort.

B. Hardware Accelerated Kalman Filtering

Hardware-based solutions to manage the computational complexity of the EKF have previously been proposed. An early effort [11] offered substantial speedup over software, but required multiple FPGAs to implement. More recently an alternative implementation of the EKF [1] has been proposed for enhanced numerical properties. There has also been proposed a specialized FPGA architecture to support Kalman Filtering for SLAM applications normally processed on a higher power processor such as a Pentium M [6].

Another, more application-agnostic group of implementations employ architectures based on the systolic array. The systolic array is a well-known structure which, when implemented on hardware, can support substantial parallelism while reducing signal fanout and other common bottlenecks to design scaling. It has been recognized for some time as an efficient basis for a hardware-based implementation for matrix math, and subsequently as a means to accelerate the Kalman Filter, since at least the early 90's [12]. In addition to a few other architectural approaches, the work in [12] describes the Fadeev algorithm, which performs modified Gauss-Jordan elimination on an input block matrix, M, comprised of four specially-chosen submatrices, **A**,**B**,**C**, and **D**. The bold typeface has been applied to the submatrix names to differentiate them from those used for state-space notation.

$$M_{2n\times 2n} = \left(\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline \mathbf{C} & \mathbf{D} \end{array}\right) \quad (3)$$

In Equation 3 (and throughout this paper), n is the number of states in the specific state-space model of plant behavior. The work in [7] extends [12] by providing extensive FPGA implementation details, as well as describing a folding approach to reduce spacial complexity (at the expense of increasing time complexity). The approach is partially motivated by the need to maintain high power efficiency for battery-powered applications. The work in [8] adds even more array folding, essentially decoupling the required number of math operations from the required number of processing elements–but in general the implementation complexity is very high.

All of these approaches use the Fadeev algorithm to implement a general-purpose matrix algebra functional unit, and then rely on software to provide the application-specific matrices to the hardware at every step within the Kalman Filter algorithm. A recent architectural variation in [9] attempts to reduce the communication overhead to the coprocessor by introducing a hardware sequencer to automatically compute the application agnostic steps of the algorithm. However, the matrices representing the linearized model must still be sent to the coprocessor on every iteration. Furthermore, in all aforementioned works the software-based application-specific portion of the update process still requires evaluation of the full non-linear function in the prediction step (e.g. $f(\hat{x}_k, u_k)$

and $g(\hat{x}_k, u_k)$) , and the evaluation of the Jacobians of the non-linear functions, every iteration.

Our work also employs the Fadeev algorithm to implement the matrix-math portion of the Kalman Filter. However, we also study the use of a piecewise-affine system modeling approach as a means to develop a coprocessor with significantly less communication requirements compared to EKF-based approaches. For brevity we term this the Piecewise-Affine Kalman Filter (PWAKF). This enables a nearly fully-autonomous hardware-based Kalman Filter having very predictable timing behavior for control applications.

C. Piecewise Affine Model of a Plant

Piecewise affine modeling is a well-known approach to linearizing a non-linear model, as discussed in the summative works [13] and [14]. Rather than linearize a non-linear model around a particular operating point, which yields a single system of equations, it is possible to divide the non-linear space into regions, wherein all points within the region are computed via a linear (or affine) function. The goal is to obtain a much wider operational range, while making a tradeoff of computations for memory. In terms of applications, it is often used as a way to model the nonideal behavior of many engineered systems, such as tire slip in an unmanned ground vehicle [15]. Other systems may explicitly require a piecewise modeling method such as for simulation of switching circuits [16], or Kalman-filter based fault detection for switching circuits [17],which requires very high sample rates. More broadly speaking, it is also an efficient means to model "nearly" linear nonlinear systems.

One issue with piecewise affine modeling may be the difficulty in finding good partitions for large state-space models. It is worth mentioning there is existing work on automated system identification, for example [18].

We assume that non-linear models of interest can be mapped into the following generic state-space form.

$$x_{k+1} = \begin{bmatrix} x_{1|k+1} \\ x_{2|k+1} \\ \vdots \\ x_{n|k+1} \end{bmatrix} = f(x_k, u_k) = \begin{bmatrix} f_1(x_k, u_k) \\ f_2(x_k, u_k) \\ \vdots \\ f_n(x_k, u_k) \end{bmatrix} \quad (4)$$

$$y = \begin{bmatrix} y_{1|k+1} \\ y_{2|k+1} \\ \vdots \\ y_{p|k+1} \end{bmatrix} = g(x_k, u_k) = \begin{bmatrix} g_1(x_k, u_k) \\ g_2(x_k, u_k) \\ \vdots \\ g_p(x_k, u_k) \end{bmatrix} \quad (5)$$

In the general case, partitioning such a multivariate function into q linear regions will yield a set of equations of the form in Equation 6.

$$\begin{aligned} x_{k+1} &= A_i x_k + B_i u_k \\ y_k &= C_i x_k + D_i u_k \\ \forall k &= 1, 2, ..., \infty, \forall i = 1, 2, ..., q \end{aligned} \quad (6)$$

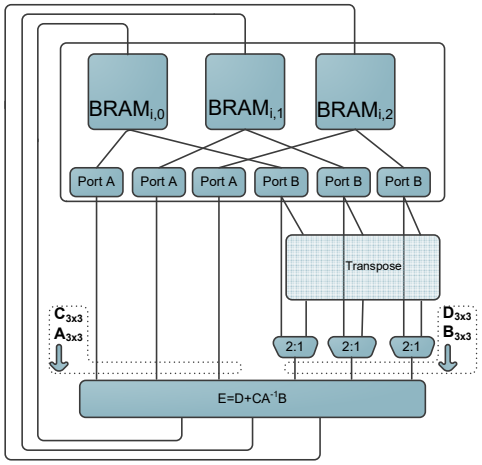

Fig. 2. Architecture datapath ($n = 3$) to support the Fadeev algorithm math unit. All routes shown have 32-bit width. The columns of block matrix M emerge from each BRAM port and enter the Fadeev unit at the bottom.

The form is based on the linear state space model. Each of q regions can each be identified by a globally unique identifier i, which we may treat like an additional hidden state. Our PWAKF approach assumes that the plant is modeled in this form.

III. System Architecture

Fig. 1 shows the general structure of the hardware. The targeted hardware platform for implementation is the Xilinx Zynq 7z020 system-on-a-chip (SoC). The Zynq SoC consists of an ARM Cortex A9 processor (termed the Processing System, or PS) coupled with an FPGA (termed the Programmable Logic, or PL). This capable processor supports the NEON SIMD instruction set, which provides acceleration for floating-point heavy applications–as much as a 50% runtime decrease was observed with it enabled. The Xilinx toolchain for the Zynq directly supports hardware-software co-design, making the platform ideal for developing co-processor based applications. FPGA components are written in VHDL and software components are written in C.

The AXI bus is a 32-bit wide standardized interface which allows a coprocessor to communicate with the PS, or allows independent communication among co-processors. As shown in Fig. 1, the coprocessor has a slave interface, which allow their internal memory spaces to appear to the PS as memory-mapped peripherals–an interface very familiar to the embedded engineer. Meanwhile, the AXI Bus master interface on the coprocessor allows it to read from sensors or other functional units within the FPGA while the system is running.

The architecture within the coprocessor is shown in Fig. 2. The regular structure of the systolic array in Fig. 3 allows us to use VHDL's *generate* and *loop* statements in such a way that the systolic array can be produced to support processing of a state-space model of arbitrary size n. This enormously simplifies the work needed to test the design at various scales.

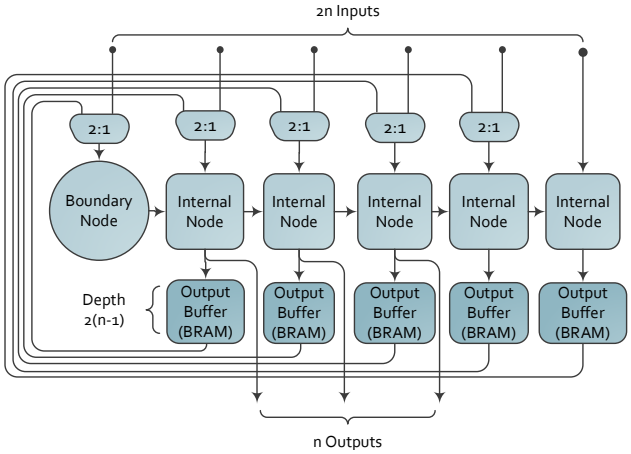

Fig. 3. Internal systolic structure of Fadeev functional unit (n=3).

A. Fadeev Algorithm

As the Fadeev algorithm has already been implemented in the previous work described in Section II-B, we only describe the approach abstractly here; more details on the theory of the algorithm appear in [19]. The algorithm is implemented as a non-homogeneous array consists of a node which performs division, which is referred to as a Boundary Node (the pivoting column in Gauss-Jordan elimination) and $2n-1$ Internal Nodes, which perform multiplication and addition (where n is the number states). The result is the Schur Complement of the **A** matrix of Equation 3. The Schur Complement is shown in Equation 7, and the hardware structure appears in Fig. 3.

$$E = \mathbf{D} + \mathbf{C}\mathbf{A}^{-1}\mathbf{B} \tag{7}$$

As an example (similar to step 5 of Table I, for instance), the following demonstrates the block matrix inputs needed to compute the matrix inverse of some matrix **A**, using identity matrix I and zero matrix **0**.

$$M = \left(\begin{array}{c|c} \mathbf{A} & I \\ \hline I & \mathbf{0} \end{array}\right) \tag{8}$$

The Fadeev algorithm then consumes input matrix M and effectively computes the Schur Complement of the **A** submatrix, which yields $\mathbf{A}^{-1}$.

$$E = 0 + I\mathbf{A}^{-1}I = \mathbf{A}^{-1} \tag{9}$$

Our work thus far uses [7] as a basis for implementing the Fadeev algorithm, as it scales better in hardware resources than a full systolic array implementation (linear vs. quadratic). The tradeoff is that runtime is a function of n^2, as inputs must be iteratively cycled through a single row of nodes rather than a full 2-dimensional array. A schedule of inputs is developed for this basic hardware element to produce a complete hardware-accelerated Kalman Filter (Tables I and II). The control block consists of a state machine which need only set the correct set of base pointers for reading and writeback during each algorithm step (Fig. 4). Therefore it is straightforward to implement different variations of the same algorithm.

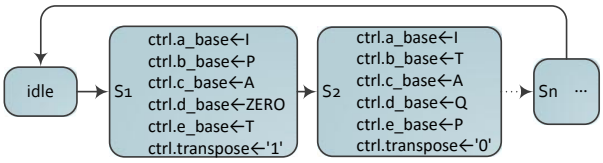

Fig. 4. The state machine for the control unit is simple and flexible enough to implement a variety of algorithms. In each state, a set of base pointers into memory are set to the appropriate constant value at that step in the algorithm.

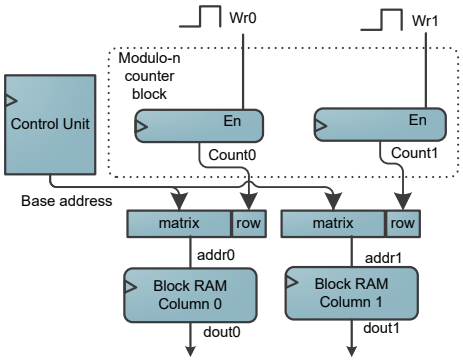

Fig. 5. Example address generation circuit ($n = 2$) for writeback. Read address generation uses an identical circuit driven by a set of "Rd" signals.

B. Memory Interface

In the worst case the Fadeev systolic array needs to write back n words simultaneously, and read $2n$ words simultaneously; thus we need to place matrix columns in separate dual-port block RAMs (BRAMs). For each BRAM, Port A is attached to the columns 0 to $n/2-1$ of the Fadeev input bus (left side), and Port B is attached to columns $n/2$ to $n-1$ (right side).

From the perspective of the PS, the global address to locate a specific element within a matrix is formed by concatenating the matrix (base) address, the matrix row address, and the matrix column address. The latter two address components are $ceil(log_2(n))$ bits wide. Columns are located in separate BRAMs, and rows are accessed in parallel by applying the same address to the n column BRAMs. Non-matrix variables (such as x, y, etc) in the PWAKF occupy the same sized memory block as any other matrix, and the unused matrix elements are simply filled with zeros.

Within the coprocessor itself, it is not necessary to specify the row column address, as it is implicit in the structure of the interconnects. The signals from the Fadeev unit signaling that the writeback data is valid drive n instances of *modulo-n* counters, which produce the row address within each BRAM. An example of address generation for writeback is shown in Fig. 5. Independent selection of rows is needed to accommodate the skewed input-output pattern of the systolic array.

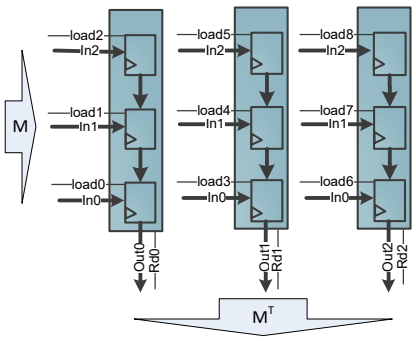

Fig. 6. Architecture of transpose functional unit ($n = 3$), comprised of parallel shift registers.

C. Matrix Transpose

As we are attempting to implement the full Kalman Filter in hardware, we need to consider how to handle the matrix transposes. The obvious solution is to pre-compute them and store them in memory. However, after partitioning the algorithm there are several temporary matrices for which the transpose cannot be pre-computed–it has to be done online.

Another option is to simply manipulate the address provided to the BRAMs so that we sequentially fetch all elements in a column of our input matrix to produce the row of our transposed matrix. This is not a scalable solution, however, since multiplexers will be needed between the BRAMs and the Fadeev input so that each BRAM output port can be re-routed to every array input column. For large n and 32-bit words, this quickly results in thousands of multiplexed connections.

Instead, we propose a simple functional consisting of shift registers to transform the rows output from the BRAMs into columns (Fig. 6). Matrix property $AB^T = (BA^T)^T$ allows us to move all transposes in the Kalman algorithm to the **B** position within our input block matrix (Equation 3). As a result, we need only provide this unit on the right-hand side datapath. Furthermore, due to the skewed input pattern characteristic of systolic arrays, inputs on the left-hand side will be consumed first, so that inputs on the right-hand side do not need to be available immediately (ie instead, n cycles after the first element on the furthest left-hand side is input). This effectively hides the extra clock latency involved in loading the transpose unit.

D. Hardware-Software Partition

We next break down the workload distribution between the PS (i.e. software) and the PL (i.e. hardware) to distinguish the EKF and PWAKF approaches.

1) EKF Approach: For the EKF, the PS needs to obtain the previous state estimate from the PL ($\hat{x}_{k-1}$), then uses it along with the non-linear state update equation $f(x_k, u_k)$ and measurement equation $g(x_k, u_k)$ to compute the following in software.

$$A^x_{k-1} = \left.\frac{\partial f(x_{k-1}, u_{k-1})}{\partial x_{k-1}}\right|_{x_{k-1}=\hat{x}^+_{k-1}} \tag{10}$$

TABLE I
INPUT SCHEDULE FOR EXTENDED KALMAN FILTER

Step	Computation	Fadeev Input (M)
1	$T = AP^T$	$\begin{bmatrix} I & P^T \\ A & 0 \end{bmatrix}$
2	$P = AT^T + Q$	$\begin{bmatrix} I & T^T \\ A & Q \end{bmatrix}$
3	$T = CP^T$	$\begin{bmatrix} I & P^T \\ C & 0 \end{bmatrix}$
4	$K = CT^T + R$	$\begin{bmatrix} I & T^T \\ C & R \end{bmatrix}$
5	$K = TK^{-1}$	$\begin{bmatrix} K & I \\ T & 0 \end{bmatrix}$
6	$P = P - KT^T$	$\begin{bmatrix} I & T^T \\ -K & P \end{bmatrix}$
7	$x = x + KE$	$\begin{bmatrix} I & E \\ K & x \end{bmatrix}$

$$\hat{x}^-_k = f(\hat{x}^+_{k-1}, u_{k-1}) \tag{11}$$

$$C^x_k = \left.\frac{\partial g(x_k, u_k)}{\partial x_k}\right|_{x_k=\hat{x}^-_k} \tag{12}$$

$$y_k = g(\hat{x}^-_k, u_k) \tag{13}$$

$$E_k = z_k - y_k \tag{14}$$

The resulting A_k and C_k matrices, along with the state estimate $\hat{x}_k$ and vector E_k need to be written back to the PL. Vector E_k from Equation 14 is the error between the sensor measurement and estimated measurement value based on the plant model. The PL sequence of computations using the Fadeev hardware are listed in Table I, and correspond to lines 6-9 in Algorithm 1. In the table, subscripts are omitted for simplicity. At the completion of this computation, the PS will read the updated state estimate from the PL in preparation for the next iteration.

2) PWAKF Approach: For the PWAKF, the PS must obtain the previous state estimate from the PL ($\hat{x}_{k-1}$). It then needs to determine the region index i within the piecewise-affine state-space that this state lies (see Equation 6). If there are relatively few regions, a simple region by region bounds check would suffice. For more complex systems, a binary search tree might be used such as in [20]. If hyperrectagular partitions are used, constant-time lookup is possible as the current region identifier can be used as an index into a multi-dimensional array.

The PS then writes to the PL a subset of elements of A_k, B_k, C_k or D_k as required by the linear model, but *only if we have entered a new region*, which reduces AXI bus communication overhead. This can be done simply by maintaining the index of the current region in software.

The PL sequence of computations using the Fadeev hardware are listed in Table II. Note the additional steps 7-10, which are the linearized computations corresponding to the non-linear function computed in software for the EKF

TABLE II
INPUT SCHEDULE FOR PIECEWISE-AFFINE KALMAN FILTER

Step	Computation	Fadeev Input (M)
1	$T = AP^T$	$\begin{bmatrix} I & P^T \\ A & 0 \end{bmatrix}$
2	$P = AT^T + Q$	$\begin{bmatrix} I & T^T \\ A & Q \end{bmatrix}$
3	$T = CP^T$	$\begin{bmatrix} I & P^T \\ C & 0 \end{bmatrix}$
4	$K = CT^T + R$	$\begin{bmatrix} I & T^T \\ C & R \end{bmatrix}$
5	$K = TK^{-1}$	$\begin{bmatrix} K & I \\ T & 0 \end{bmatrix}$
6	$P = P - KT^T$	$\begin{bmatrix} I & T^T \\ -K & P \end{bmatrix}$
7	$T = Ax$	$\begin{bmatrix} I & x \\ A & 0 \end{bmatrix}$
8	$x = T + Bu$	$\begin{bmatrix} I & u \\ B & T \end{bmatrix}$
9	$T = Cx$	$\begin{bmatrix} I & x \\ C & 0 \end{bmatrix}$
10	$y = T + Du$	$\begin{bmatrix} I & u \\ D & T \end{bmatrix}$
11	$T = z - y$	$\begin{bmatrix} I & I \\ -y & z \end{bmatrix}$
12	$x = x + KT$	$\begin{bmatrix} I & T \\ K & x \end{bmatrix}$

approach. Another difference is that the error term $z - y$ is computed in software for the EKF approach (and later transmitted over the AXI bus), but is computed in hardware (step 11) in the PWAKF approach. At the completion of these computations, the PS will read the updated state estimate from the PL in preparation for the next iteration.

IV. IMPLEMENTATION RESULTS

A. Resources

The Zynq chip has $n_{dspmax} = 220$ available DSPs, and our usage directly impacts our space-time tradeoff. These are a critical resource for floating point-heavy designs and should be used strategically. There are $2n - 1$ Internal Nodes in the array (where n is the number of states), each containing a multiplier and an adder. If we allocate n_{dsp}=4 DSPs per Internal Node, the maximum value of n is 28. If we allocate 5 DSPs per Internal Node, the maximum value of n is 21 (without increasing FF/LUT usage). Thus to ensure all nodes use only DSPs, we must ensure $n_{dsp}(2n-1) \leq n_{dspmax}$.

The system resources were assessed at various sizes of n in Fig. 7. The difference between the EKF and PWAKF approach lies only in the controller steps; thus the impact on LUT usage is negligible. Resource growth is primarily linear in nature: LUT growth is a linear function of n, BRAM growth is $2n$, DSP growth is $n_{dsp}(n-1)$. The transpose unit consists of an array of registers and grows as a factor of n^2.

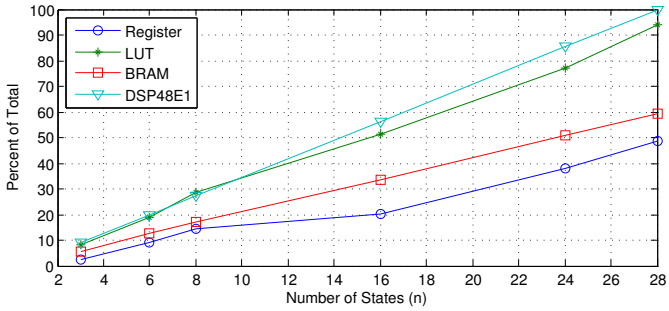

Fig. 7. Architecture resource consumption for various values of n. Scaling halts at $n = 28$ due to exhaustion of DSP units, although there are additional SoCs in the Zynq line with significantly more DSPs. Additional PEs would be implemented in reconfigurable logic, and would be quickly exhausted.

B. Performance

The performance of the PWAKF approach is compared to that of the EKF approach: first at the hardware level, which makes for easier comparison across implementations, and then at the mixed hardware-software level, for which most existing literature presents highly application-specific results. Software-based EKF benchmarks use GNU Scientific Library– a well-known library with optimized matrix algebra routines– and use *gcc*'s flag $-O2$, which activates the NEON floating point instructions. The FPGA implementations are clocked at 45Mhz, while the ARM processor is operated at 200Mhz which lies in the spectrum of microcontroller frequencies used in embedded systems [1].

1) Hardware Level: A performance comparison of hardware vs. software appears in Table III. The PWAKF coprocessor shows less speedup compared to the EKF one, because it performs several additional computation steps which are fairly trivial in relation to the time complexity of the Fadeev unit. Still, at $n = 21$, the PWAKF implementation updates in $490\mu s$ and the EKF in $286\mu s$, both of which are notably faster than the hardware-only implementation ($782.92\mu s$) reported for the UD filter in [1]. Of interest is the elevated speedup around $n = 2ton = 4$ both in Table III and Fig. 9. This is due to the fact that the problem size was too small for the PS to efficiently map into NEON instructions.

The number of clock cycles for a complete algorithm iteration can be analytically expressed as a function of the latency of the divider, multiplier, and adder units. This allows us to determine the number of clocks needed for various values of n, shown in Equation 15.

$$k = (D_{ma}(2((n^2) - 1)) + D_{ma}(2n - 2))K_s + D_d \quad (15)$$

The relevant symbols are as follows: D_{ma} is the clocks needed to complete either multiplication or addition (they are the same), D_d is the clocks needed to complete division, n is the number of states, and K_s is the number of steps in the Kalman algorithm.

2) Mixed Hardware-Software Level: The advantage of the PWAKF approach does not become apparent without analyz-

TABLE III
HARDWARE-ONLY VS. SOFTWARE-ONLY UPDATE TIME

n	EKF SW 200Mhz Time (ms)	PWAKF HW (45Mhz) Time (ms)	PWAKF HW (45Mhz) Speedup	EKF HW (45Mhz) Time (ms)	EKF HW (45Mhz) Speedup
2	0.073	0.011	6.70	0.006	10.10
4	0.069	0.019	3.57	0.011	5.46
6	0.156	0.043	3.64	0.025	5.82
8	0.258	0.075	3.45	0.044	5.52
10	0.430	0.115	3.73	0.067	6.12
12	0.692	0.164	4.21	0.096	6.95
14	0.985	0.222	4.44	0.130	7.40
16	1.396	0.288	4.85	0.168	8.11
18	2.020	0.363	5.57	0.212	9.33
20	2.681	0.446	6.01	0.260	10.11
22	3.422	0.538	6.36	0.314	10.74
24	4.264	0.638	6.68	0.372	11.27
26	5.283	0.747	7.07	0.436	11.92
28	6.409	0.864	7.42	0.504	12.51

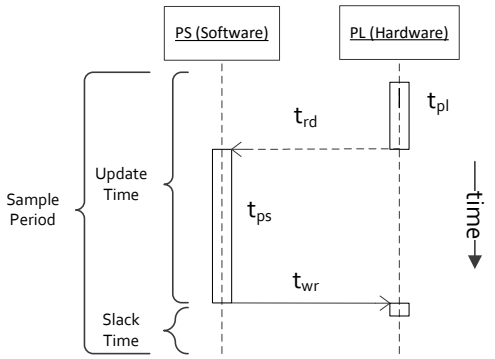

Fig. 8. Sequence Diagram showing the set of delays considered in this study. With this configuration, the update timing is driven by the hardware. The software prepares the matrix values needed for the next iteration.

ing the mixed hardware-software performance. In this section we analyze how the PWAKF can provide a speedup over the EKF, while also shifting the majority of the computation (greater than 99%) to timing-deterministic hardware.

The general idea is that the PWAKF approach will present less communication overhead and less overhead on the application-specific part of the computation in software, because these processes need only occur when the state of the plant under observation migrates into a new state-space region. For example, a plant which is operating in steady-state may incur no matrix-update communication overhead at all.

Since the performance of both the EKF and PWAKF approaches depend on the characteristics of the application, in order to make generalized analysis, we introduce some abstract application parameters. Fig. 8 summarizes the sources of processing delay which are considered during this analysis, and analysis parameters are described in Table IV.

As the number of states n needed in the plant model increases, so does the hardware-software communication time (t_{rd}+t_{wr}), the software processing time (t_{ps}) and the hardware processing time (t_{pl}). For the EKF approach, the communication time is rather predictable: during each update, the PS will read the n-word state vector from the PL, and write back the $n^2 + nm + p$ words of A,C, and E.

For the PWAKF approach, the communication time depends on the number of elements in the linear state-space model which are not constant across regions. This ranges from 0 (a trivial case in which the plant model was already linear), to $n(n+m) + p(n+m)$, meaning the entire model (A,B,C and D) needs to be transmitted during every region transition. This latter case is quite unlikely as it requires every state variable in the model to appear in all state update equations, all output equations, *and* be involved in a non-linear operation. For example, consider the non-linear state update equation for a system consisting of two states, and arbitrary constants c_x.

$$x_{k+1} = f(x_k, u_k) = \begin{bmatrix} c_1 x_1^2 + c_2 x_2^2 \\ c_3 x_1 + c_4 x_2 + u \end{bmatrix} \tag{16}$$

Based on Equation 6, the non-linear model can be expressed as a piecewise affine model with q regions, shown below.

$$x_{k+1} = \begin{bmatrix} m_{i,1} & m_{i,2} \\ c_3 & c_4 \end{bmatrix} x + \begin{bmatrix} b_{i,1} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ u \end{bmatrix} \quad \forall i = 1, 2, ..., q \tag{17}$$

Slope ($m_{i,x}$) and offset ($b_{i,x}$) elements are introduced, and the vector containing the single input u has been augmented with a constant 1 to accommodate the offset. Specific values are determined after partitioning and system identification; for example a simple point-to-point procedure is described in [21]. During a region transition, only these entries need to be updated in the PL memory, as they differ among regions.

For analysis, we use Equation 16 as a template for our non-linear plant model, and modulate the "complexity" of the model by adding additional non-linear (quadratic) terms based on parameter r_{nc}, which ranges from 0 to 1. For example, if r_{nc}=0.5, this indicates half of all elements in the linear model are variables and therefore must be transfered during a region transition. Plant model profiles based on this template consist of Low Complexity (r_{nc}=0.1), Moderate Complexity (r_{nc}=0.5), and High Complexity (r_{nc}=0.9).

Since the *rate* of region transition varies by application, we also introduce a transition rate parameter r_t which varies from 0 (i.e. there are no region transitions) to 1 (i.e. a region transition occurs every timestep). Finally, overall mixed hardware-software speedup is assessed using Equation 18.

$$speedup = \frac{t_{swb}}{t_{pl} + t_{ps} + t_{rdb} n + t_{wrb} r_t n_{wb}} \tag{18}$$

We further define n_{wr} (number of words to write back to the PL) as shown in Equation 19.

$$n_{wb} = \lfloor (n(n+m) + p(n+m)) r_{nc} \rfloor \tag{19}$$

Analysis results based on Equation 18 and the application profiles are summarized in Table V. To manage the number of variables, we assume $m = p = 1$, which indicates a single-input single-output system. The speedup of the hardware-software PWAKF in the best case ($r_t = r_{nc} = 0$) and worst

TABLE IV
PERFORMANCE ANALYSIS PARAMETERS

Symbol	Description
n	Number of states in linear state-space model
m	Number of control inputs in linear state-space model
p	Number of measurable outputs in linear state-space model
r_{nc}	Rate of non-constant entries in the linear state-space model
r_t	Plant region transition rate (transitions per unit timestep)
t_{ps}	Time spent on the software processor (PS)
t_{pl}	Time spent on the hardware processor (PL)
t_{rd}	Time spent by the PS reading from the PL
t_{wr}	Time spent by the PS writing to the PL
t_{rdb}	Benchmarked time for the PS to read a word from the PL
t_{wrb}	Benchmarked time for the PS to write a word to the PL
t_{swb}	Benchmarked software-only implementation execution time

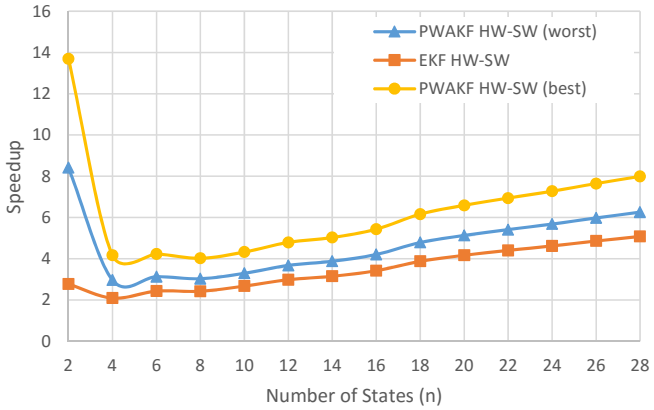

Fig. 9. Mixed hardware-software speedup over software vs. number of states: PWAKF shows an average 62% performance increase over hardware-software EKF. The large speedup for small n is due to unavailability of NEON instructions at that problem size, highlighting the high impact that these instructions have on software execution time.

case ($r_t = r_{nc} = 1$) is compared to the hardware-software EKF in Fig. 9. On average, the speedup for the PWAKF HW-SW approach is 62% larger than the EKF HW-SW approach.

3) Discussion: Although in the hardware-only implementation the PWAKF does not offer as large a speedup as the EKF approach when compared to the equivalent software-only implementation (Table III), when the timing of a *complete application* is considered, the PWAKF approach outperforms the EKF approach due to the reduced time spent in the PS and on communication. At the hardware level, speedup for the hardware-software PWAKF could be further increased by reconsidering steps 7-10 of the PWAKF algorithm, which are rather trivial with respect to the time complexity of the Fadeev algorithm and would be significantly faster if computed in a separate functional unit. This approach was used in [9], reporting an additional 62% performance increase.

Theoretically, for the EKF linearization occurs at runtime (during each discrete time step), while for the PWAKF the function is linearized offline and stored in state-space form. Accurate state tracking for the EKF depends (in part) on the time step being sufficiently small to support the assumption that the plant evolves linearly within the time step; likewise for the PWAKF, good tracking behavior depends on the regions being sufficiently small to assume linearity within that region of the plant's state-space. Therefore, EKF may be preferable if the number of regions needed to achieve the desired accuracy for the PWAKF method induces substantial memory overhead, such as for systems with very fast oscillations. As a straightforward enhancement to the PWAKF, introducing a small amount of hysteresis (e.g. overlap) in the region boundaries could further reduce overhead for noisy or oscillatory systems.

In terms of speedup, it is evident larger problems will benefit more than smaller ones. However, for any application, time-deterministic computations allow a control engineer to focus on plant dynamics rather than worry about (and compensate for) non-idealities in the computational platform.

C. Power Consumption

The Xilinx Power Estimator tool, combined with signal activity rates obtained by simulating our largest hardware design ($n = 28$), estimates the Zynq chip to consume 722mW overall, including dynamic and static power. Of this, the coprocessor (PL) alone consumes 149mW. Therefore the approach is power-efficient and is suitable for many embedded systems with limited power budgets.

V. CONCLUSION

The PWAKF coprocessor is a novel approach to hardware acceleration for Kalman-filter state estimation, establishing a new reference point in the mixed hardware-software design continuum. By replacing the standard EKF methodology with a fully linearized one, it offers a speedup over both the pure software approach, as well as the hardware-software EKF approach, without sacrificing the modeling expressiveness that software enjoys. There are several avenues for further exploration. Placing constraints on the model partitioning scheme (e.g. hypercubes [20]) may enable the region-identification task to be placed in hardware. Of additional interest is the possibility to integrate the Kalman Filter with prior work in FPGA-based LQR control [10] to form more advanced FPGA-based controllers. The ability to develop a complete system on an FPGA brings with it the promise of high-speed, low-power, tightly integrated control systems.

REFERENCES

[1] R. Gonzalez, G. Sutter *et al.*, "FPGA-based floating-point UD filter coprocessor for integrated navigation systems," in *Argentine Conference on Embedded Systems (CASE)*, Aug 2015, pp. 7–12.

[2] E. Monmasson and M. Cirstea, "Guest Editorial Special Section on Industrial Control Applications of FPGAs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1250–1252, Aug 2013.

[3] S. Kozak, "Advanced control engineering methods in modern technological applications," in *Carpathian Control Conference (ICCC)*, May 2012, pp. 392–397.

[4] E. Monmasson, L. Idkhajine *et al.*, "FPGA-based Controllers," *IEEE Industrial Electronics Magazine*, vol. 5, no. 1, pp. 14–26, March 2011.

[5] B. Mutlu and M. Dolen, "Implementations of state-space controllers using Field Programmable Gate Arrays," in *International Symposium on Power Electronics Electrical Drives Automation and Motion (SPEEDAM)*, June 2010, pp. 1436–1441.

TABLE V
PWAKF HW-SW SPEEDUP VS. SOFTWARE-ONLY APPROACH, WITH VARYING MODEL CHARACTERISTICS

	Low Complexity (r_{nc}=0.1)					Moderate Complexity (r_{nc}=0.5)					High Complexity (r_{nc}=0.9)				
n \ r_t	0	0.25	0.5	0.75	1	0	0.25	0.5	0.75	1	0	0.25	0.5	0.75	1
2	13.70	13.32	12.96	12.62	12.30	13.70	12.62	11.70	10.90	10.20	13.70	11.99	10.66	9.59	8.72
4	4.17	4.09	4.00	3.92	3.85	4.17	3.95	3.75	3.57	3.41	4.17	3.82	3.53	3.28	3.06
6	4.23	4.17	4.10	4.04	3.98	4.23	4.04	3.86	3.70	3.55	4.23	3.92	3.65	3.41	3.21
8	4.03	3.97	3.91	3.86	3.80	4.03	3.86	3.70	3.56	3.42	4.03	3.75	3.51	3.30	3.11
10	4.33	4.27	4.22	4.17	4.11	4.33	4.16	4.00	3.85	3.71	4.33	4.05	3.80	3.58	3.38
12	4.80	4.74	4.69	4.64	4.58	4.80	4.61	4.44	4.28	4.13	4.80	4.49	4.22	3.98	3.76
14	5.03	4.98	4.92	4.87	4.82	5.03	4.84	4.67	4.50	4.35	5.03	4.72	4.44	4.19	3.97
16	5.44	5.38	5.33	5.27	5.22	5.44	5.24	5.05	4.88	4.72	5.44	5.10	4.80	4.54	4.30
18	6.17	6.11	6.04	5.98	5.93	6.17	5.94	5.74	5.54	5.36	6.17	5.79	5.46	5.16	4.90
20	6.60	6.53	6.47	6.41	6.34	6.60	6.36	6.14	5.93	5.74	6.60	6.20	5.84	5.53	5.25
22	6.95	6.88	6.82	6.75	6.69	6.95	6.70	6.47	6.26	6.06	6.95	6.53	6.16	5.83	5.53
24	7.28	7.21	7.15	7.08	7.02	7.28	7.02	6.79	6.56	6.36	7.28	6.85	6.46	6.12	5.81
26	7.65	7.58	7.51	7.45	7.38	7.65	7.38	7.14	6.90	6.69	7.65	7.20	6.79	6.43	6.11
28	8.00	7.93	7.86	7.79	7.72	8.00	7.72	7.46	7.22	7.00	8.00	7.53	7.11	6.73	6.40

[6] V. Bonato, E. Marques *et al.*, "A Floating-Point Extended Kalman Filter Implementation for Autonomous Mobile Robots," in *International Conference on Field Programmable Logic and Applications*, Aug 2007, pp. 576–579.

[7] A. Bigdeli, M. Biglari-Abhari *et al.*, "A New Pipelined Systolic Array-based Architecture for Matrix Inversion in FPGAs with Kalman Filter Case Study," *EURASIP J. Appl. Signal Process.*, vol. 2006, pp. 75–75, Jan. 2006.

[8] A. Sudarsanam, "Analysis of Field Programmable Gate Array-Based Kalman Filter Architectures," in *All Graduate Theses and Dissertations*, 2010. [Online]. Available: http://digitalcommons.usu.edu/etd/788

[9] D. Pritsker, "Hybrid implementation of Extended Kalman Filter on an FPGA," in *IEEE Radar Conference (RadarCon)*, May 2015, pp. 0077–0082.

[10] A. Mills, P. Zhang *et al.*, "A software configurable coprocessor-based state-space controller," in *International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–6.

[11] C. Lee and Z. Salcic, "A fully-hardware-type maximum-parallel architecture for Kalman tracking filter in FPGAs," in *International Conference on Information, Communications and Signal Processing (ICICS)*, Sep 1997, pp. 1243–1247 vol.2.

[12] F. Gaston and G. Irwin, "Systolic Kalman filtering: an overview," *Control Theory and Applications*, vol. 137, no. 4, pp. 235–244, Jul 1990.

[13] M. Johansson, *Piecewise Linear Control Systems: A Computational Approach*. Springer, 2002.

[14] J. Xu and L. Xie, *Control and Estimation of Piecewise Affine Systems*, ser. Woodhead Publishing Reviews Mechanical Engineering. Elsevier Science & Technology, 2014.

[15] A. Benine-Neto and C. Grand, "Piecewise affine control for fast unmanned ground vehicles," in *International Conference on Intelligent Robots and Systems (IROS)*, Oct 2012, pp. 3673–3678.

[16] P.-K. Luk, S. Aldhaher *et al.*, "State-Space Modeling of a Class $\mathbf{e^2}$ Converter for Inductive links," *IEEE Transactions on Power Electronics*, vol. 30, no. 6, pp. 3242–3251, June 2015.

[17] H. Al-Sheikh, G. Hoblos *et al.*, "Piecewise switched Kalman filtering for sensor fault diagnosis in a DC/DC power converter," in *Technological Advances in Electrical, International Conference on Electronics and Computer Engineering (TAEECE)*, April 2015, pp. 62–67.

[18] A. Buchan, D. Haldane *et al.*, "Automatic identification of dynamic piecewise affine models for a running robot," in *International Conference on Intelligent Robots and Systems (IROS)*, Nov 2013, pp. 5600–5607.

[19] M. Zajc, R. Sernec *et al.*, "An efficient linear algebra SoC design: implementation considerations," in *Mediterranean Electrotechnical Conference (MELECON)*, 2002, pp. 322–326.

[20] F. Comaschi, B. Genuit *et al.*, "FPGA Implementations of Piecewise Affine Functions Based on Multi-Resolution Hyperrectangular Partitions," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, no. 12, pp. 2920–2933, Dec 2012.

[21] G. Lowe and M. Zohdy, "A technique for using H_2 and H_∞; robust state estimation on nonlinear systems," in *International Conference on Electro/Information Technology (EIT)*, June 2009, pp. 109–115.