A Software Configurable and Parallelized Coprocessor Architecture for LQR Control

Pei Zhang, Aaron Mills, Joseph Zambreno, and Phillip H. Jones
Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA 50010
Email: {peizhang,ajmills,zambreno,phjones}@iastate.edu

***Abstract*—We present a software configurable and parallelized coprocessor architecture for Linear Quadratic Regulator (LQR) control that can control physical processes representable by a linear state-space model. Our proposed architecture has distinct advantages over purely software or purely hardware approaches. It differs from other hardware controllers in that it is not hardwired to control one or a small range of plant types (e.g. only electric motors). Via software, an embedded systems engineer can easily reconfigure the controller to suit a wide range of control applications that can be represented as a state-space model. One goal of our approach is to support a design methodology to help bridge the gap between control and embedded system software engineering. Control of the well-understood inverted pendulum on a cart is used as an illustrative example of how the proposed hardware accelerator architecture supports our envisioned design methodology for helping bridge this gap. Additionally, we explore the design space of our co-processor's parallel architecture in terms of computing speed and resource utilization. Our performance results show a 3.4 to 100 factor speedup over a 666 MHz embedded ARM processor, for plants that can be represented by 4 to 128 states respectively.**

***Index Terms*—FPGA, state space control, LQR, co-processor, parallel processing.**

I. Introduction

Cyber Physical Systems (CPS) can be thought of as systems that have or require a tight coupling between their computing and physical aspects. We assert that advancing science, technology, and engineering to manage and exploit this tight coupling will require improved interaction between researchers from computing domains (e.g. software and hardware engineering), and physical domains (e.g. control engineering), and that field-programmable gate arrays (FPGAs) can act as a medium to help bridge gaps between these research fields.

FPGAs are of growing interest in the area of applied control theory [1]. In addition to the massive parallelism available on FPGAs that can potentially 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 control concerns (e.g. accounting for update-rate jitter).

Efficient implementation of a control algorithm on an FPGA can be challenging for engineers unfamiliar with hardware architecture design. One solution is software programmable hardware. In our work, we describe a software-configurable FPGA co-processor architecture that can implement a wide range of linear state-space controllers, up to the complexity of a Linear Quadratic Regulator (LQR) coupled with a Luenberger Observer [2]. For the purpose of evaluation, the controller can be interfaced to a hardware-based emulation of a physical plant using what we will refer to as a Plant-on-Chip (PoC). This arrangement is depicted in Fig. 1. The PoC allows for rapid and consistent testing of control algorithms and system platform configurations [3]. Once stability of the emulated plant is achieved, it can be replaced with an interface to the actual plant's sensors and actuators. All control computations are done in hardware, while software running on the CPU is used to initialize the co-processor. The software is also free to perform other actives: task scheduling, path planning, video processing, or interactive communications.

The big picture usage model for our software configurable co-processor based controller is that control engineers would focus on the mathematics of their controller, without the concern of computing artifacts breaking assumptions, such as deterministic sample rates or the representable range of numbers. Meanwhile, embedded system engineers would focus on making efficient utilization of CPU resources, without the concern of stringent timing constraints often associated with controlling physical plants. Control engineers would interact with embedded systems engineers by providing the appropriate state-space matrices require to program the co-processor.

Contributions. The three primary contributions of this paper are: 1) the implementation of a software-configurable LQR co-processor using single-precision floating point arithmetic that helps bridge the gap between control and embedded system engineering, 2) the design space exploration of the parallel architecture proposed, and 3) a transformation of the standard LQR control algorithm to make it better suited for hardware implementation.

Organization. The remainder of this paper is organized as follows. In Section II, we discuss and compare related works in this problem space. Section III gives a brief introduction to the concept of state-space modeling and the LQR control algorithm. This section also describes a transformation for converting the standard representation of the LQR controller into a form the is better suited for hardware implementation. In Section IV, we describe the detailed design of our software configurable parallelized co-processor architecture. Section V presents an illustrative example of using our co-processor to evaluate the use of hardware verses software for an embedded control application, and explores the performance and scaling of our coprocessor-based controller. Section VI concludes this paper and provides avenues of future work.

978-1-4673-9406-2/15/$31.00 ©2015 IEEE

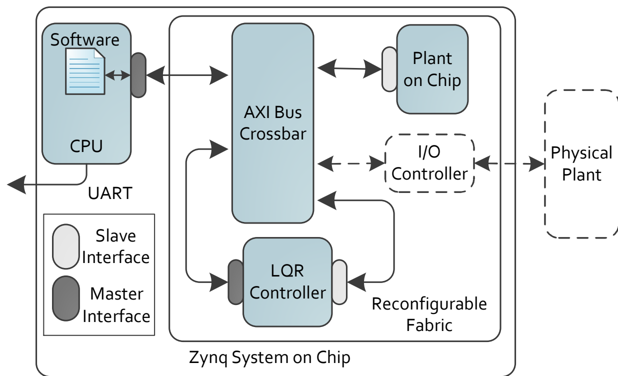

Fig. 1: System Overview. An example system-level organization where the controller and Plant-on-Chip (PoC) are fully software configurable.

II. Related Work

Kozak, in [4], surveys trends in the field of applied control, in which we see control has evolved from manually-tuned single-input single-output (SISO) controllers (e.g. Proportion Integral Derivative (PID) control) to multiple-input multiple output (MIMO) controllers (e.g. H_∞ control and model-predictive controller (MPC)). The latter types of algorithms are computationally intense and can introduce significant latencies when implemented with off-the-shelf processing platforms. Kozak additionally suggests that a software-hardware co-design approach for implementing advanced controllers (e.g. H_∞) in FPGAs would enable designers to make better use of these complex controllers in high-speed systems. Monmasson, in [5], makes a similar suggestion, pointing out how different parts of a control algorithm are better suited for different types of hardware. However, locating the optimal software-hardware partition is still a challenge.

There are numerous examples of application-specific FPGA-based controllers in the literature. An example of a system requiring very fast control update rates appears in [6], in which a high-speed pan/tilt camera is designed to track objects. In order to reach the 3.5ms update rate, a dedicated PC is used to perform image processing and produce motor control signals. It is noted that the PC introduced considerable delay in the feedback loop. Another application requiring very high update rates appears in [7], which presents an application-specific design that used machine vision to control an inverted pendulum. In [8], the authors developed a self-tuning state-space controller using a multiply-accumulate unit which is interfaced with a digital signal processor (DSP). The use of FPGAs to control a plant with non-constant plant parameters was demonstrated. In [9], the design of a high-speed, hardware-only, fixed-point MPC is discussed. Finally, in [10], an MPC is implemented on an FPGA and is shown to allow for significantly faster sample rates than a PC running at a higher clock frequency.

Compared to software, implementing high-performance control algorithms in hardware is relatively time consuming and often leads to application-specific solutions. A proposed solution to this issue appears in [11] and [12], which use a co-processor to perform low-level repetitive matrix operations for MPC. This allows control designers to use software for the high-level logic; however, to do so they had to work with a custom floating-point format and instruction set.

A general summary of approaches used to implement controllers on FPGAs appears in [13]. In addition, a call is made for designs that make efficient use of the massive parallelism available on FPGAs, while retaining the generality and flexibility available to software solutions. Our work pursues this goal. A number of works exist describing controllers that achieve reduced computational delay. However, these controllers are designed to solve specific problems, unlike our fully software configurable solution. Garbergs, in [14], [15], presents the closest work to our approach. The main differences are: 1) their design is highly sequential and does not explore parallelization of a state-space based controller, 2) their design does not consider scaling to different sized controllers (e.g. if controller coefficients change the design must be re-implemented), 3) their design is intended to be standalone, as compared to being a memory-mapped co-processor, and 4) their vision focuses more on developing a fast hardware controller as opposed to supporting a design methodology that bridges the gap between embedded systems and control engineers.

III. LQR Control Algorithm

This section first gives a brief overview of state-space modeling. Next, the "standard" form of the LQR control algorithm is provided. A transformation of this standard form is then presented, and a discussion is given that illustrates the computing advantages associated with using the transformed formulation of the LQR algorithm.

A. State Space Modeling

A discrete state-space model defines what state a system will be in one time step into the future, in terms of the current state of the system and current input acting upon it. A generic linearized discrete state-space system model consists of matrices A, B, C, and D[1] and is formulated as follows:

$$x_{k+1} = Ax_k + Bu_k \tag{1}$$

$$y_k = Cx_k \tag{2}$$

Where:

- x_k represents the state of the system at time k
- u_k represents the input acting on the system at time k
- y_k represents outputs of the system at time k
- A is an $N \times N$ matrix that defines the internal dynamics of the system
- B is an $N \times M$ matrix that defines how the input impacts system state
- C is a $P \times N$ matrix that transforms states of the system into outputs (y_k)

[1] it is common to omit the matrix D, as inputs typically do not directly impact output

Equation 1 is referred to as the state update equation. With respect to a closed loop control system, matrix A represents the dynamics of the plant being controlled, matrix B represents how actuator commands (i.e. u_k) impact the plant, and the matrix C could be viewed as a mapping of the current state to the output obtained from sensors (i.e. y_k).

Also the width (i.e. number of columns) of each matrix or length of each vector found in Equation 1 and 2 can be viewed as follows:

- M: the number of system/plant inputs/actuators.
- N: the number of system/plant states.
- P: the number of system outputs (i.e. sensors).

B. Linear-quadratic Regulator (LQR)

An LQR controller makes use of a gain matrix K, which specifies a linear combination of plant states that are used as feedback when computing plant control values (u_k). A particular K is sought such that the feedback law, Equation 5, minimizes the cost function given by Equation 3 [16].

$$J(u) = \sum_{1}^{\infty} x_k^T Q x_k + u_k^T R u_k \tag{3}$$

Generating K requires a control engineer to tune a state-cost matrix (Q) and a performance index matrix (R) in a manner that causes system behavior to fulfill specific application requirements (e.g. actuator energy expended, rise-time, settling-time). Since K is derived systematically to minimize the cost function $J(u)$, once Q and R have been tuned, it is referred to as an optimal controller. Derivation of the gain matrix K is beyond the scope of this paper, but it is a fairly straightforward process.

Given a gain matrix K, Equations 4 - 6 specify what we will call the "standard" form of the LQR control algorithm.

$$\hat{x}_{k^+} = G\big(y_k - C\hat{x}_k\big) + \hat{x}_k \tag{4}$$

$$u_k = -K\hat{x}_{k^+} \tag{5}$$

$$\hat{x}_{k+1} = A\hat{x}_{k^+} + Bu_k \tag{6}$$

Where:

- $\hat{x}_k$ is an $N \times 1$ estimated state vector
- $\hat{x}_{k^+}$ is an $N \times 1$ estimated state vector after correction
- G is an $N \times P$ observer matrix
- K is an $M \times N$ feedback gain matrix

These equations can be viewed as performing the following tasks: 1) predicting the state of the plant in the next time step, shown in Equation 6, 2) correcting this prediction based on sensor information, shown in Equation 4, and 3) computing the control values (u_k) to send to the plant, shown in Equation 5. The prediction and correction steps are generally referred to as the observer portion of the controller. Specifically in our case, Equations 6 and 4 define a Luenberger type Observer.

C. LQR Algorithm Transformation for Hardware Design

The "standard" form of the LQR algorithm (Equations 4 - 6) is convenient for gaining intuition for how the algorithm works. However, it is not convenient from a computation and hardware architecture perspective. The primary issue with directly implementing the standard from is the data dependence that occurs between Equation 4 and 5 (i.e. $\hat{x}_{k^+}$ is required for computing u_k). This data dependency can be removed by manipulating the standard form to obtain Equation 7. Note, this alternative eliminates $\hat{x}_{k^+}$. Next, Equations 7 and 8, can be simplified to Equation 9, where T is a constant matrix defined by Equation 10. This allows the observer prediction/correction and control computation to be represented by a single matrix-vector multiplication.

$$u_k = -KGy_k + (KGC - K)\hat{x}_k \tag{7}$$

$$\hat{x}_{k+1} = (A-BK)Gy_k + \big((A-BK)-(A-BK)GC\big)\hat{x}_k \tag{8}$$

$$\begin{bmatrix} u_k \\ \hat{x}_{k+1} \end{bmatrix} = T \begin{bmatrix} y_k \\ \hat{x}_k \end{bmatrix} \tag{9}$$

where:

$$T = \begin{bmatrix} -KG & KGC - K \\ (A-BK)G & (A-BK)-(A-BK)GC \end{bmatrix} \tag{10}$$

is an $M + N$ by $N + P$ matrix.

In addition to removing the $\hat{x}_{k^+}$ data dependence, the transformed version of the algorithm has other advantages from a computation and hardware architecture design perspective. First, if the standard form was directly implemented, then additional storage would be required for the intermediate $\hat{x}_{k^+}$ values and additional on-chip communication bandwidth would be required to move these intermediate values between computing and storage units. Second, direct implementation would either a) require separate resources for each of the equations implemented, or b) would require relatively complex control logic to allow computing resources to be shared, while the alternative form allows leveraging existing research in the area of parallelizing matrix-vector multiplication.

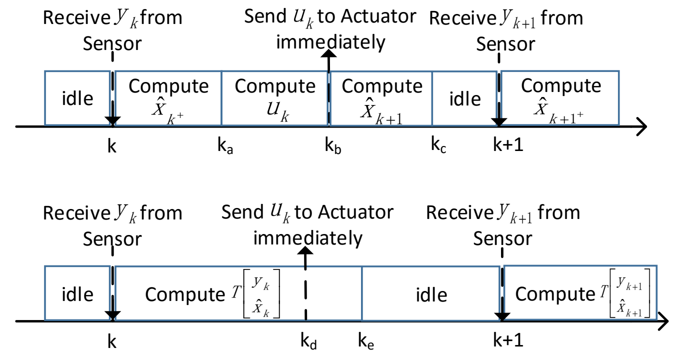

Fig. 2: Operating Schedule for Current Observer Based LQR. The top one shows timing for the original algorithm and the bottom one shows the reorganized timing. Subscripted k denotes discrete time points between algorithm steps k and $k+1$.

Figure 2 qualitatively illustrates the difference in relative computation time using the "standard" form verses the transformed version of the LQR algorithm. As can be seen for the standard form, $\hat{x}_{k+}$ must be computed before computing u_k. Also note, while in both cases the control command (u_k) can be sent to the plant before computing the next prediction, the control command and prediction computations complete earlier for the alternative form than for the standard form.

Figure 3 quantitatively shows the difference in clock cycles to compute the LQR algorithm using the standard form verses the alternative form. It is assumed in both cases a hardware architecture is used that can compute one matrix-vector product per clock cycle (assuming no data dependencies); this is described in Section IV. It is also assumed that the dimensions of the matrices and vectors are N=M=P (where N, M, and P were defined earlier). Intuitively, the two primary issues with efficiency when computing using the standard form as compared to the transformed version are: 1) the $\hat{x}_{k+}$ data dependency causes pipeline stalls, and 2) the time required to compute $\hat{x}_{k+}$. The number of clock cycles to compute the control vector (u_k), and to compute an iteration of the algorithm (i.e. time to compute u_k and x_{k+1}) is given by: $N + M - 2 + T_p(G) + T_p(K)$ and $2N + M - 3 + T_p(G) + T_p(K) + T_p(AB)$ respectively. The time for these same values for the transformed version is given by $M - 1 + T_p(T)$ and $M + N - 1 + T_p(T)$ respectively.

The function $T_p(number\ of\ columns\ in\ matrix)$ used above represents the time for the pipeline of an architecture to be filled while processing a matrix. It is a function of the number of floating point adders and multipliers used as well as the latency of these components, and is proportional to the number of columns contained by the matrix being processed. T_p is derived in greater detail in Section IV.

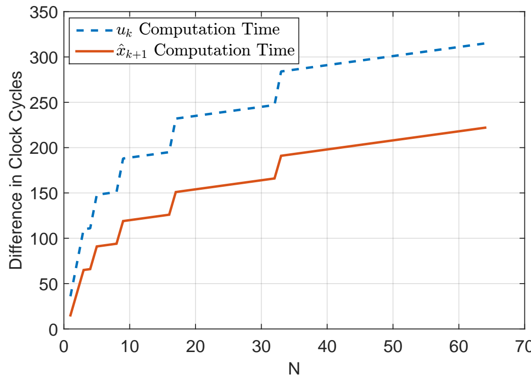

Fig. 3: Clock cycles saved in computing u_k (blue) and one whole state update (red) when comparing the reorganized formula with the original. The reduction in clock cycles increases as the number of states increase.

Two important implications of this analysis are that the transformed algorithm will: 1) have a shorter "Control Delay" (i.e. time between receiving new sensor values and sending a control command) than the standard form, and 2) support faster sensor update rates, since the time to compute the algorithm using this form is less than when using the standard form.

IV. Architecture

This section presents the architecture of our software configurable LQR coprocessor. A high-level overview of the coprocessor is given, followed by a description of each of its main components.

A. Overview

Figure 4 depicts the architecture of our co-processor. It is composed of three major parts: 1) a multiply- accumulate parallel processing architecture, 2) matrix/vector storage, and 3) configuration registers for parameters. Software sets the configuration registers with the number of states (N) used to model the plant being controlled, the number control commands (M) to compute, the number of sensor values (P) to receive from the plant, the depth of the multiply-accumulate engine, and the adder/multiplier latency. Next, matrix/vector storage is configured with the T matrix provided by the control engineer. The multiply-accumulate engine ("Parallel Processing Architecture") then runs the LQR control algorithm.

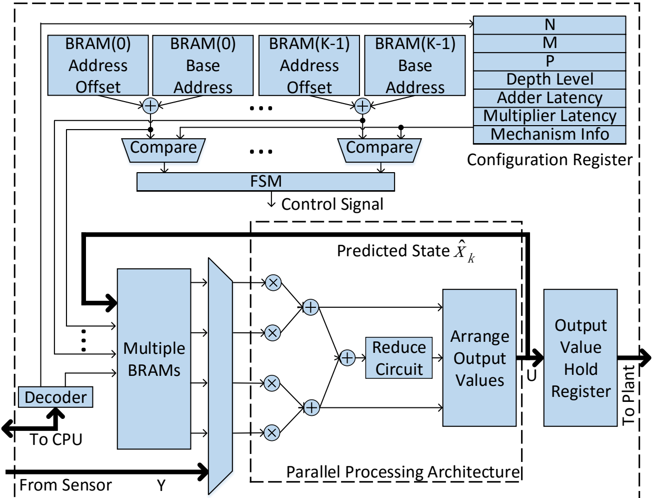

Fig. 4: Architecture of the controller's main compute unit.

B. Parallel Processing Architecture

The parallel processing architecture implemented is based closely on the architecture presented in [17] for parallelizing sparse-matrix vector multiplication for large matrices. Since our T matrix will not typically be sparse, we have been able to simplify our control logic for scheduling multiply accumulate operations onto our binary tree structure. Just as the architecture presented in [17], our architecture is composed of three aspects: 1) a binary tree structure for parallel computation of multiply-accumulation operations, 2) taps for allowing the tree structure to process multiple rows of a matrix in parallel (referred to as *merging*), and 3) a *reduction* mechanism to allow computation of matrix rows that have more columns than the tree has multipliers.

Binary Tree Structure. Figure 5 illustrates the binary tree structure that allows processing multiply-accumulate operations in parallel. Assuming the number of columns in the T matrix is equal to the number of multiplier leaf nodes of the tree, then a new matrix row- vector dot product can enter the tree's pipeline each clock cycle. We call the number of adder levels of the tree its depth. If we denote the latency of a multiplier as L_M and the latency of an adder as L_A, then the total latency of the binary tree structure is $L_{total} = L_M + depth \times L_A$.

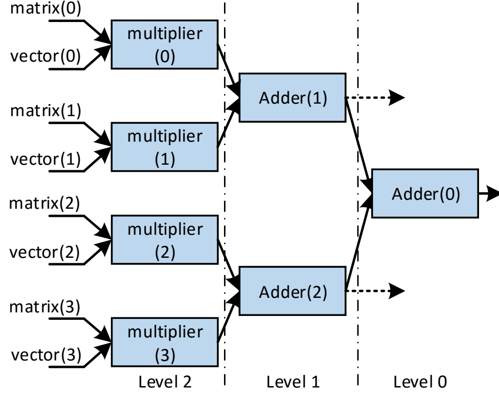

Fig. 5: Tree-based multiply-accumulate architecture (depth=2)

Merge. When the number of multipliers in the multiply-accumulate tree is at lease twice the number of columns in the T matrix, then the tree can be split in half and outputs can be tapped from the nodes at level one (as shown in Figure 5). This allows two rows of the T matrix to start processing each clock cycle. After a latency of $L_{total} = L_M+(depth-1)\times L_A$, corresponding results are output from Adder(1) and Adder(0) at level 1, thus doubling the output rate of the coprocessor. We refer to this as the *merge* feature of this architecture.

In fact, we can merge more than two rows of matrix T into the tree each clock cycle. Suppose the T matrix has c columns and the binary tree architecture has depth equal to D_p. In the simplified case, if $N_f = 2^{D_p - \lceil \log_2(c) \rceil}$, then N_f rows of matrix T can be feed into the tree structure each clock cycle.

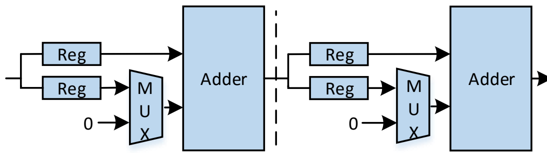

Fig. 6: Reduction circuit structure

Reduce. When the number of columns in the T matrix is larger than the number of multipliers in the multiply-accumulate tree, then each row has to be feed into the tree structure over multiple clock cycles. We will let N_g represent the number of clocks needed to read in a row. Given that the number of columns in T is equal to $N+P$, $N_g = \lceil (N+P)/j \rceil$, where j is the number of multipliers. The circuit shown in Figure 6 is called a reduction circuit and was introduced by [18]. Its purpose is to sum the results of the N_g iterations needed to process a row of T.

The number of such reduction components required is $N_g - 1$. The reduction circuit allows correct operation when the number of rows in T is larger than the number of multipliers in the tree, at the cost of latency. The latency of the reduction circuit for processing a row of T is given by $(L_A+2)\times(N_g-1)$. A matrix T consisting of c columns and l rows would require the processing time given in Equation 11.

$$T_{BR} = L_M + L_A D_p + (L_A+1)(N_g-1) + (l-1)N_g \quad (11)$$

Resource Analysis. Table I lists the configurable parameters of the coprocessor. It defines how to compute the number of adders required for both the reduction circuit and the binary tree circuit, as a function of these parameters. The "Latency-Clocks per Row" entry in the table indicates the clock cycles required from entering the multiplier to reaching the output of the multiply-accumulate tree.

TABLE I: Summary of Architecture Parameters

Merge Mechanism (Same as normal mode when $\lceil log_2(c) \rceil = D_p$)	
Configurable Paramater	Column:c Depth:D_p
Number of Rows per Fetch, N_f	$2^{D_p - \lceil \log_2(c) \rceil}$
Multipliers in Binary Tree	2^{D_p}
Adders in Binary Tree	$2^{D_p} - 1$
Latency–Clocks per Fetch	$L_M + \lceil \log_2(c) \rceil L_A$
Reduce Mechanism (Same as normal mode when $\lceil c/2^{D_p} \rceil = 1$)	
Configurable Paramater	Column:c Depth:D_p
Number of Groups per Row, N_g	$\lceil c/2^{D_p} \rceil$
Multipliers in Binary Tree	2^{D_p}
Adders in Binary Tree	$2^{D_p} - 1$
Adders in Reduction Circuit	$\lceil c/2^{D_p} \rceil - 1$
Latency–Clocks per Row	$L_M + D_p L_A + (\lceil c/2^{D_p} \rceil - 1)(L_A + 2)$

C. Matrix/Vector Storage

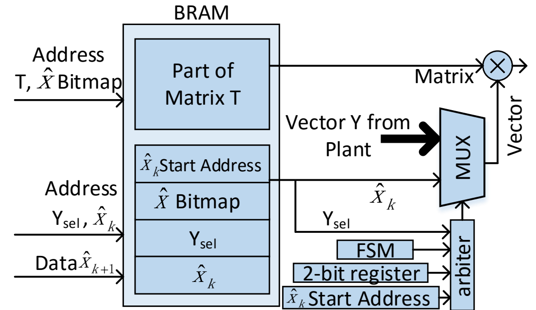

Fig. 7: Communication between BRAM and Multiplier

In our design, matrix T is stored in block RAMs (BRAMs) as shown in Figure 7. Algorithm 1 defines how software must store T into each BRAM. Each multiplier in our designed is associated with an individual BRAM; Figure 7 illustrates how each BRAM connects to its associated multiplier.

Algorithm 1 Memory Map algorithm for T Matrix

```
                    ▷ T is an l by c matrix and binary tree structure depth is D_p
procedure MERGE MECHANISM MEMORY MANAGEMENT
    N_f = 2^(D_p − ⌈log_2(c)⌉)                         ▷ Number of rows merged
    for m = 0 to ⌈l/N_f⌉ − 1 do
        for k = 0 to N_f − 1 do
            for n = 0 to 2^(D_p)/N_f − 1 do
                if n ≤ c − 1 then
                    BRAM(n + k2^(D_p)/N_f) ⇐ T_(k+mN_f, n)
                else
                    BRAM(n + k2^(D_p)/N_f) ⇐ 0
                end if
            end for
        end for
    end for
end procedure
```

```
procedure REDUCE MECHANISM MEMORY MANAGEMENT
    for m = 0 to l − 1 do
        for k = 0 to ⌈c/2^(D_p)⌉ − 1 do
            for n = 0 to 2^(D_p) − 1 do
                if n + k2^(D_p) ≤ c − 1 then
                    BRAM(n) ⇐ T_(m, n+k2^(D_p))
                else
                    BRAM(n) ⇐ 0
                end if
            end for
        end for
    end for
end procedure
```

V. Hardware Implementation and Analysis

A. Evaluation Setup

Our software configurable LQR coprocessor was prototyped using a Zedboard, which hosts a Xilinx Zynq FPGA (XC7Z020). The Zynq FPGA is composed of a processing system (PS), which has a dual-core ARM Cortex-A9 processor that can run at up to 666 MHz, and a programmable logic (PL) fabric for deploying custom hardware designs. The LQR coprocessor was instantiated in the PL, and used a 100 MHz clock frequency for all tested configurations of the controller.

Three sets of evaluation experiments were performed. First, the LQR controller was used to control a PoC emulating a pendulum on a cart, and compared against results obtained from Matlab to verify the correctness of our implementation. Second, performance experiments were performed to evaluate the computing time of systems with T matrices having $N+P=8$ columns upto $N+P=256$ columns, and for coprocessors having 4 to 64 multipliers in their multiply-accumulate tree structure. It should be noted Table IV gives entries for 128 and 256 multipliers (i.e. depths 7 and 8); however, these are analytically computed. They are provided to give a sense for the largest coprocessor that could be implemented if we had the largest available Zynq FPGA (XC7Z100). The third experiment measured the computing time of the LQR algorithm running in software on the ARM processor, for system matrix sizes (T) having $N+P=8$ to $N+P=64$ columns.

Figure 8 depicts the top-level system structure and the direct connection between the controller and the PoC, which enables parallel loading of sensor values. It can be contrasted with Figure 1, which requires sequential loading of sensor values over the AXI bus.

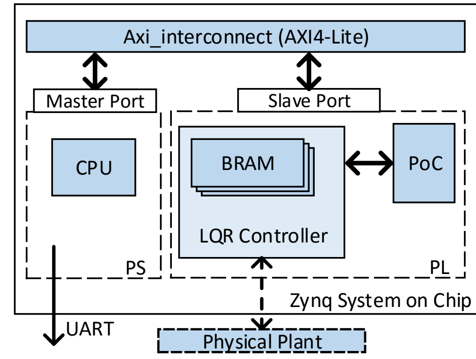

Fig. 8: Top level view of the experimental platform, highlighting the direct connection between controller and PoC.

B. Controller Evaluation using a Plant-on-Chip

A Plant-on-Chip (PoC) was deployed onto the FPGA's PL fabric to emulate the pendulum-on-a-cart plant shown in Figure 9, using the model parameters given in Table II. The PoC executes the state space Equations 1 and 2, with input u_k received from the hardware or software-based controller. The state of the PoC and control commands are logged out of band using an UART interface, which allowed unobtrusively plotting controller behavior. This model requires the CPU to configure the co-processor parameters as $N=4$, $M=1$, and $P=2$. The state vector consists of four states: x, $\dot{x}$, ϕ, and $\dot{\phi}$ (see Figure 9). The pendulum was initially configured with a starting position of $-5°$ from vertical. When running the hardware controller at 100 MHz, it could compute u_k in 530ns, and complete a full iteration of the algorithm in 600ns. The hardware control graph is shown in Figure 10, and was found to match the results obtained from Matlab.

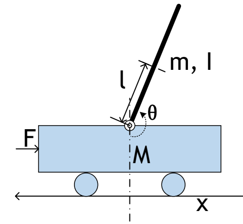

Fig. 9: Inverted Pendulumn Model

TABLE II: Inverted Pendulum Model Parameters

Symbol	Meaning	Initialization
M	cart mass	2.725kg
m	pendulum mass	1.09kg
b	coefficient of friction	0.1 N/m/sec
l	length to pendulum center of mass	0.2 m
I	pendulum moment of inertia	$0.006 kg \cdot m_2$
F	applied force	0
x	position displacement	0
θ	angle from vertical	$-5°$

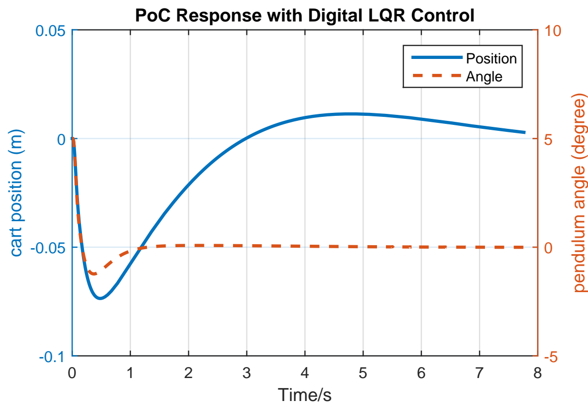

Fig. 10: Pendulum Control Plot

TABLE IV: Hardware Computation Time (μs) at 100MHz. Assume N=M=P for each system size.

Depth \ System Size		N=4	8	16	32	64	128
2	$k-k_d$	0.62	1.10	2.54	7.03	24.62	89.9
	$k-k_e$	0.73	1.45	3.85	12.18	45.13	171.85
3	$k-k_d$	0.58	0.82	1.54	3.94	12.58	45.22
	$k-k_e$	0.65	1.01	2.21	6.53	22.85	86.21
4	$k-k_d$	0.56	0.74	1.10	2.30	6.62	22.94
	$k-k_e$	0.63	0.85	1.45	3.61	11.77	43.45
5	$k-k_d$	0.55	0.70	0.94	1.54	3.70	11.86
	$k-k_e$	0.62	0.81	1.13	2.21	6.29	22.13
6	$k-k_d$	0.55	0.68	0.86	1.22	2.30	6.38
	$k-k_e$	0.62	0.79	1.05	1.57	3.61	11.53
7	$k-k_d$	\	0.67	0.82	1.06	1.66	3.70
	$k-k_e$	\	0.78	1.01	1.41	2.33	6.29
8	$k-k_d$	\	0.67	0.80	0.98	1.34	2.42
	$k-k_e$	\	0.78	0.99	1.33	2.01	3.73

C. Hardware verses Software

Table III and IV summarizes the results obtained when comparing the computing time of the software controller running on the ARM processor (at 100 MHz, 333 MHz, and 666 MHz), against the hardware controller running at 100 MHz. The expression $k - k_d$ represents the time from when the controller receives new sensor data to the time it completes computation of u_k, and $k - k_e$ represents the time from when the controller receives new sensor data to completing an iteration of the control algorithm (See Figure 2). The results show that the hardware controller achieves about a 3.4 to 100 factor speedup over the embedded ARM processor, for plants that can be represented by $N = 4$ to $N = 128$ states respectively. The high end of our speedup (100x) is reasonable, given that the highest performing configuration we can fit on our FPGA utilizes 64 multipliers in parallel, with a deeply pipelined 6-level multiply-accumulate tree. Note that data in Table IV with a shaded background is when the number of rows in the T matrix equals the number multipliers used by the coprocessor. Data points above these cells make use of the *reduction* mechanism, and data points below these cells make use of the *merge* mechanism.

TABLE III: Software Computation Time (μs). Assume N=M=P for each system size.

Clock \ System Size		N=4	8	16	32
100MHz	$k-k_d$	11.08	36.93	136.12	528.06
	$k-k_e$	22.56	73.84	272.91	1056.93
333MHz	$k-k_d$	3.327	11.09	40.877	158.577
	$k-k_e$	6.775	22.174	81.955	317.396
666MHz	$k-k_d$	1.664	5.545	20.438	79.288
	$k-k_e$	3.387	11.087	40.977	158.698

D. Resource Utilization

Table V summarizes the hardware resource utilization and maximum obtainable clock frequency as the parallelism of the coprocessor is increased from having 8 to 64 parallel multipliers in its multiply-accumulate tree. Given that we are explicitly pairing a BRAM with each multiplier, the BRAMs scale as expected. The DSP also scale as expected, given that each floating point multiplication makes use of two DSP blocks, and floating point addition units make use of LUT-based cores. Also as expected, the obtainable clock frequency decreases with increased coprocessor size. However, this is a modest decrease, given the rate of increase in resources. The resource usage appears to be fairly well balanced, though a number of LUTs could be freed up if a number of floating point adders were made to be DSP-based.

TABLE V: Hardware Resource Usage

System Size (N=M=P)	Flip-Flops (106400 total)	LUTs (53200 total)	Minimum BRAM (140 total)	DSP48E (220 total)	Maximum Frequency
4	5793	5137	10	16	153.657MHz
8	12065	10376	18	32	132.749MHz
16	24426	21077	34	64	127.356MHz
32	48143	42138	66	128	122.205MHz

VI. Conclusion

In support of improving the performance of future Cyber Physical Systems (CPS), a software configurable and parallelized LQR co-processor architecture was presented to help bridge the gap between control and embedded system engineers. A transformation was also given for converting the "standard" form of an LQR algorithm into an alternative form that was better suited for hardware parallelization. Our performance results show a 3.4 to 100 factor speedup over a 666 MHz embedded ARM processor, for plants that can be represented by 4 to 128 states respectively. Two avenues of future work include: 1) developing a more formalized procedure for moving controller designs from theory to implementation, and 2) exploring the support of higher complexity controllers, such as, H_∞ and Model Predictive Control (MPC).

References

[1] E. Monmasson and M. Cirstea, "Guest editorial special section on industrial control applications of fpgas," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 3, pp. 1250–1252, Aug 2013.

[2] D. Luenberger, "An introduction to observers," *IEEE Transactions on Automatic Control*, vol. 16, no. 6, pp. 596–602, Dec 1971.

[3] S. Vyas, N. Chetan Kumar, J. Zambreno, C. Gill, R. Cytron, and P. Jones, "An fpga-based plant-on-chip platform for cyber-physical system analysis," *Embedded Systems Letters, IEEE*, vol. 6, no. 1, pp. 4–7, March 2014.

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

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

[6] K. Okumura, H. Oku, and M. Ishikawa, "High-Speed Gaze Controller for Millisecond-Order Pan/Tilt Camera," in *IEEE International Conference on Robotics and Automation*, May 2011, pp. 6186–6191.

[7] Y. Tu and M. Ho, "Design and implementation of robust visual servoing control of an inverted pendulum with an FPGA-based image co-processor," *Mechatronics*, vol. 21, no. 7, pp. 1170 – 1182, 2011.

[8] D. A. Gwaltney, K. D. King, K. J. Smith, and J. Montenegro, "Implementation of Adaptive Digital Controllers on Programmable Logic Devices," *Military and Aerospace Programmable Logic Devices (MAPLD)*, Sept 2002.

[9] K. Basterretxea and K. Benkrid, "Embedded high-speed model predictive controller on a FPGA," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2011.

[10] J. Jerez, G. Constantinides, and E. Kerrigan, "An FPGA implementation of a sparse quadratic programming solver for constrained predictive control," in *In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, June 2011, pp. 209–218.

[11] P. Vouzis, M. Kothare, L. Bleris, and M. Arnold, "A System-on-a-Chip Implementation for Embedded Real-Time Model Predictive Control," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1006–1017, Sept 2009.

[12] L. Bleris, P. Vouzis, M. Arnold, and M. Kothare, "A co-processor FPGA platform for the implementation of real-time model predictive control," in *American Control Conference*, June 2006.

[13] 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.

[14] B. Garbergs and B. Sohlberg, "Specialised hardware for state space control of a dynamic process," in *TENCON '96. Proceedings., 1996 IEEE TENCON. Digital Signal Processing Applications*, vol. 2, Nov 1996, pp. 895–899 vol.2.

[15] ——, "Implementation of a state space controller in a fpga," in *Electrotechnical Conference, 1998. MELECON 98., 9th Mediterranean*, vol. 1, May 1998, pp. 566–569 vol.1.

[16] T. Chen, B. Francis, and T. Hagiwara, "Optimal sampled-data control systems," *Proceedings of the IEEE*, vol. 86, no. 4, pp. 741–741, 1998.

[17] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 63–74.

[18] L. Zhuo, G. Morris, and V. Prasanna, "Designing scalable fpga-based reduction circuits using pipelined floating-point cores," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005, pp. 147a–147a.