A Scalable Low-Power Reconfigurable Accelerator for Action-Dependent Heuristic Dynamic Programming

Nan Zheng, *Student Member, IEEE*, and Pinaki Mazumder, *Fellow, IEEE*

***Abstract*— Adaptive dynamic programming (ADP) is an effective algorithm that has been successfully deployed in various control tasks. For many emerging applications where power consumption is a major design consideration, the conventional way of implementing ADP as software executing on a general-purpose processor is not sufficient. This paper proposes a scalable and low-power hardware architecture for implementing one of the most popular forms of ADP called action-dependent heuristic dynamic programming. Different from most machine-learning accelerators that mainly focus on the inference operation, the proposed architecture is also designed for energy-efficient learning, considering the highly iterative and interactive nature of the ADP algorithm. In addition, a virtual update technique is proposed to speed up the computation and to improve the energy efficiency of the accelerators. Two design examples are presented to demonstrate the proposed algorithm and architecture. Compared with the software approach running on a general-purpose processor, the accelerator operating at 175 MHz achieves 270 times improvement in computational time while consuming merely 25 mW power. Furthermore, it is demonstrated that the proposed virtual update algorithm can effectively boost the energy efficiency of the accelerator. Improvements up to 1.64 times are observed in the benchmark tasks employed.**

***Index Terms*— Adaptive dynamic programming, neural networks, low-power accelerators, action-dependent heuristic dynamic programming, machine learning.**

I. Introduction

ADAPTIVE dynamic programming (ADP) is a powerful algorithm in solving various decision-making and control problems [1]–[5]. Through approximating the solution to the Bellman equation, the ADP algorithm can generate optimal or near-optimal solutions for many real-life applications. The ADP algorithm is considered one type of reinforcement-learning algorithm. It is also known as adaptive critic design, approximate dynamic programming, neurodynamic programming, etc. Many ADP algorithms have been successfully implemented in the form of software running on a general-purpose processor [6]–[19]. Among various types of ADP algorithms, the action-dependent heuristic dynamic programming (ADHDP) algorithm is one of the most popular and most powerful ADP algorithms [6], [7], [13], [14], [16], [18], as this algorithm does not require any pre-knowledge about the model of the system to be controlled.

Despite being effective as an algorithm itself, the highly iterative ADP algorithms running on a general-purpose processor in the form of software fail to provide energy-efficient solutions to various applications where power consumption is of importance. For example, potential applications for the ADP algorithm are mobile autonomous robots with a small form factor [20]–[22] and future internet of things (IoT) devices. For these microrobots and IoT devices that chiefly rely on energy scavenging from the environment or energy stored on a tiny battery, energy consumption is of utmost importance. Therefore, it is necessary to resort to specialized accelerators in order to meet the stringent requirements of both the speed and energy consumption.

In recent years, many specialized accelerators have been developed for neural networks [23]–[35]. Most of these accelerators are for deep neural networks, especially the deep convolutional neural network (CNN), which is one of the most popular and powerful neural networks employed widely in image and audio recognition. Many design techniques have been proposed to improve efficiency and throughput of the neural network accelerators. Scalable architectures are presented in [23] and [24] with an emphasis on memory optimization. Low-power techniques are proposed in [25] through leveraging the row-stationary technique so that the number of data movements can be minimized. An automated optimization-based co-design method is investigated in [26], which yields a significant improvement in energy efficiency. In [27], sparsity in the data is exploited in order to skip ineffective computation, which improves both the speed and power efficiency. A systolic structure was utilized in [34] to save the computational energy and silicon area. In [33], a novel Cartesian product-based computation architecture was introduced to effectively exploit the sparsity in a compressed-sparse CNN. Even though an ADP-based reinforcement learning also utilizes neural networks, there are some different design challenges and tradeoffs in building accelerators for ADP algorithms [36]. For example, most existing accelerators

Manuscript received July 1, 2017; revised September 18, 2017 and October 26, 2017; accepted October 27, 2017. Date of publication November 28, 2017; date of current version May 8, 2018. This work was supported by the National Science Foundation under Grant CCF 1421467 and Grant 1710940. This paper was recommended by Associate Editor G. Masera. *(Corresponding author: Nan Zheng.)*

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48104 USA (e-mail: zhengn@umich.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2017.2771437

only implement the inference phase, as the learning is assumed to be accomplished somewhere else. Such an operating model indeed works well for supervised learning. The slow and energy-consuming learning process can be conducted on the graphic processing units in the data centers. Users of the neural network accelerator can then download the trained weights onto the chip, and the accelerator is ready to conduct some classification or inference tasks. On the other hand, most ADP algorithms target controlling plants or making optimal decisions in a dynamic environment. Under this circumstance, each ADP accelerator needs to learn how to choose the optimal policy for the plant or environment it is interacting with in an on-line fashion. Therefore, learning for an ADP accelerator is most likely to be a real-time task.

In this paper, we introduce a hardware architecture as well as design methodologies for ADHDP accelerators. A tile-based computing is employed to provide good scalability for the accelerators. The designed accelerators are also flexible, as they can be programmed with instructions in order to run ADHDP algorithms with different configurations. Low-power operations are achieved through reducing the data movements by utilizing and partitioning data buffers. Furthermore, as we focus on building accelerators that can conduct learning efficiently, a virtual update technique is introduced to leverage some unique computational patterns in the ADHDP algorithm in order to shorten the computational time and increase the energy efficiency.

In the following, we provide a brief introduction of the ADHDP algorithm in Section II. The hardware architecture for the algorithm and design strategies are presented in Section III. The architecture is upgraded with the proposed virtual update algorithm in Section IV. Design examples and simulation results are provided in Section V. Section VI concludes this work.

II. Action-Dependent Heuristic Dynamic Programming

In this section, a few concepts for the ADP and ADHDP are reviewed. Only important terminologies that are closely related to this paper are covered. The interested reader is referred to the excellent review papers on this topic [1]–[5].

A. Actor-Critic Networks

Suppose the discrete-time system under control can be modeled by

$$\mathbf{x}(t+1) = f\left[\mathbf{x}(t), \mathbf{a}(t)\right] \tag{1}$$

where $\mathbf{x}(t)$ is the n-dimensional state vector at time t, $\mathbf{a}(t)$ is the m-dimensional action vector, and $f(\cdot)$ is the model of the system. The target of the algorithm is to maximize the reward-to-go J, expressed as follows

$$J\left[\mathbf{x}(t)\right] = \sum_{k=1}^{\infty} \gamma^{k-1} r\left[\mathbf{x}(t+k)\right] \tag{2}$$

where γ is the discount factor used to promote the reward received in the near future over long-term reward and $r\left[\mathbf{x}(t)\right]$ is the reward received at state $\mathbf{x}(t)$.

Equation (2) can be maximized through solving the Bellman equation

$$J^*\left[\mathbf{x}(t)\right] = \max_{\mathbf{a}(t)} \left\{ r\left[\mathbf{x}(t+1)\right] + \gamma\, J^*\left[\mathbf{x}(t+1)\right] \right\} \tag{3}$$

where $J^*\left[\mathbf{x}(t)\right]$ denotes the optimal value function under the optimal policy. The optimal policy $\mathbf{a}^*(t)$ is obtained by maximizing the right-hand side of (3).

Solving the Bellman equation directly is intractable for many problems with practical sizes. The complexity grows exponentially with the size of the problem, which is well known as the curse of the dimensionality. To circumvent this difficulty, the ADP algorithm solves the Bellman equation approximately with the help of function approximators. There are two categories of ADP algorithms: model-based ADP and model-free ADP. The model-based ADP algorithm assumes the model for the plant that is under control or the environment that the agent is interacting with is known. This type of algorithm explicitly exploits the model information in the process of solving the Bellman equation. In contrast, the model-free ADP algorithm does not need a model for the plant or the environment. The algorithm learns the model in the process of interacting with the plant or the environment. Therefore, the model-free ADP algorithm is more general and more powerful for many practical problems.

The model-free ADP algorithm we consider in this paper is the ADHDP algorithm, which is one of the most popular model-free ADP algorithm [6], [7], [13], [14], [16], [18]. It is closely related to the well-known Q-learning algorithm that is widely used by researchers in the artificial intelligence community. The configuration of the ADHDP algorithm is illustrated in Fig. 1. In the figure, $\mathbf{x}$ and $\mathbf{a}$ represent the state and action vectors, respectively. $\mathbf{h^a}$ and $\mathbf{h^c}$ are N_{ha}-dimensional and N_{hc}-dimensional output vectors from the hidden units in the actor and critic network, respectively. $\mathbf{w^{a1}}$, $\mathbf{w^{a2}}$, $\mathbf{w^{c1}}$, and $\mathbf{w^{c2}}$ are synaptic weights in the networks. Two neural networks are used as universal function approximators in this algorithm. One neural network, called critic network, is employed to generate $\hat{J}\left[\mathbf{x}(t)\right]$, which is an estimation of $J\left[\mathbf{x}(t)\right]$. The critic network attempts to learn $J\left[\mathbf{x}(t)\right]$ through adjusting the synaptic weights in the neural network in order to minimize the absolute value of the temporal difference error

$$\delta(t) = \hat{J}\left[\mathbf{x}(t-1)\right] - \gamma\, \hat{J}\left[\mathbf{x}(t)\right] - r[\mathbf{x}(t)] \tag{4}$$

The second neural network is called an actor network. Its function is to generate an action vector $\mathbf{a}(t)$ that maximizes the estimated reward-to-go $\hat{J}\left[\mathbf{x}(t)\right]$. Action vector outputted by the actor network is fed to the critic network. The actor then adjusts its synaptic weights to maximize $\hat{J}\left[\mathbf{x}(t)\right]$.

B. On-Line Learning Algorithm

In the learning process, we need to train the two neural networks such that the defined cost function can be minimized. The most popular and efficient way to train a neural network is the stochastic gradient descent learning based on backpropagation [37]. Errors at output layers are propagated back to each synapse in the network, layer by layer. There are two phases in the ADHDP algorithm: critic update phase and actor

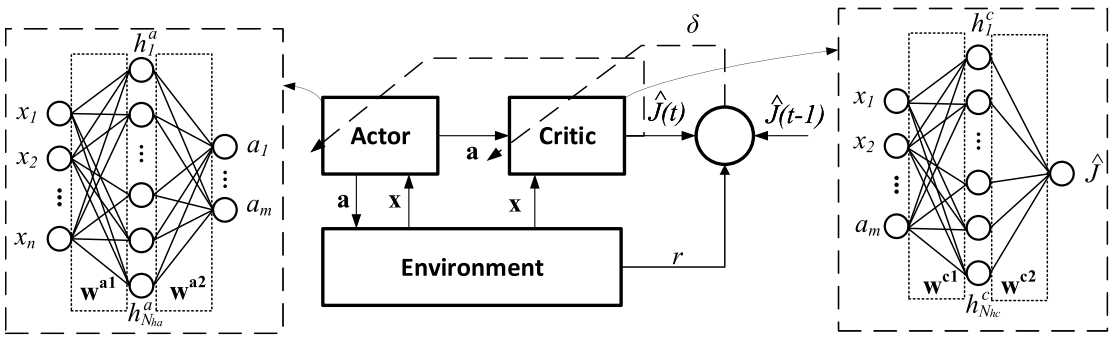

Fig. 1. Illustration of the actor-critic configurations used in the ADHDP algorithm. Two neural networks, critic network and actor network, are employed in the algorithm to approximate functions need to learn.

update phase. Multiple iterations are involved in both phases. Each iteration contains a forward operation and a backward operation. The algorithm is illustrated in Fig. 2.

1) Forward Opeartion: For each iteration in the actor update phase, the forward operation is carried out according to (5)-(9).

$$h_i^a = \sigma\left(\sum_{j=1}^{n} w_{ij}^{a1} x_j\right) \tag{5}$$

$$a_i = \sigma\left(\sum_{j=1}^{N_{ha}} w_{ij}^{a2} h_j^a\right) \tag{6}$$

$$\mathbf{p^c} = \begin{bmatrix}\mathbf{a}\\ \mathbf{x}\end{bmatrix} \tag{7}$$

$$h_i^c = \sigma\left(\sum_{j=1}^{m+n} w_{ij}^{c1} p_j^c\right) \tag{8}$$

$$\hat{J} = \sum_{i=1}^{N_{hc}} w_i^{c2} h_i^c \tag{9}$$

where $\sigma(\cdot)$ is the activation function. Popular choices are hyperbolic tangent function, sigmoid function, and rectified linear unit. For the critic forward phase, only (8)-(9) are carried out. (5)-(7) are not necessary as the weights in the actor network remain the same, which leads to the same action vector.

2) Backward Opeartion: During the backward operation in the critic update phase, $\mathbf{w^{c1}}$ and $\mathbf{w^{c2}}$ are updated according to (10) and (11).

$$\Delta w_i^{c2} = \alpha \delta h_i^c \tag{10}$$

$$\Delta w_{ij}^{c1} = \alpha e_i^{c1} \sigma'\left(\sum_{k=1}^{m+n} w_{ik}^{c1} p_k^c\right) p_j^c \tag{11}$$

where $e_i^{c1} = \delta w_i^{c2}$ is the error at the hidden unit h_i^c, and α is the learning rate.

During the backward operation in the actor update phase, $\mathbf{w^{a1}}$ and $\mathbf{w^{a2}}$ are updated according to (12) and (13).

$$\Delta w_{ij}^{a2} = \alpha e_i^{a2} \sigma'\left(\sum_{k=1}^{N_{ha}} w_{ik}^{a2} h_k^a\right) h_j^a \tag{12}$$

$$\Delta w_{ij}^{a1} = \alpha e_i^{a1} \sigma'\left(\sum_{k=1}^{n} w_{ik}^{a1} x_k\right) x_j \tag{13}$$

where $e_j^{a1} = \sum_{i=1}^{m}\left[e_i^{a2} \cdot \sigma'\left(\sum_{k=1}^{N_{ha}} w_{ik}^{a2} h_k^a\right) \cdot w_{ij}^{a2}\right]$, $e_j^{a2} = \sum_{i=1}^{N_{hc}}\left[e_i^{c1} \cdot \sigma'\left(\sum_{k=1}^{m+n} w_{ik}^{c1} p_k^c\right) \cdot w_{ij}^{c1}\right]$, and $e_j^{c1} = e_a w_j^{c2}$ are backpropagated errors at h_j^a, a_j and h_j^c, respectively. $e_a^2/2$ is the cost function that needs to be minimized for the actor network. In many applications, the desired reward-to-go is 0, i.e. no punishment (negative reward). In this case, a convenient choice is $e_a = \hat{J}\,[\mathbf{x}(t)]$ [6], [7], [13], [14], [16], [18].

III. Hardware Architecture

The proposed hardware architecture for the ADP accelerator is shown in Fig. 3. The accelerator consists of three major blocks: datapath, memory, and controller. The datapath is the core of the ADP accelerator. It handles all the arithmetic operations needed in the ADHDP algorithm. The memory unit contains all the on-chip storage units, including static random-access memory (SRAM) array used to store synaptic weights, registers for holding neuron states, and input buffers for reducing data movements. The controller oversees operations of the whole accelerator, and executes pre-programmed instructions in order.

A. On-Chip Memory

Memory in our system can be divided into three categories based on the purposes they serve: synapse memory, neuron memory, and data buffers. The most critical and also the largest memory block is the synapse memory, as the number of synapses grows quadratically with the size of the neural network. In this design, we use an SRAM array for storing synaptic weights. Neuron memory is where the activation levels of neurons in the network are stored. It is implemented with an array of registers in this design. Data buffers are storage units used to hold the input, intermediate, and output data temporarily in order to accelerate the computation and save computational energy.

Data flow and memory access patterns employed in the proposed accelerator are shown in Fig. 4. The computations in the forward operations shown in (5) - (9) are mostly matrix

Inputs : $\mathbf{w^{a1}}$, $\mathbf{w^{a2}}$, $\mathbf{w^{c1}}$, $\mathbf{w^{c2}}$: weights for the actor and critic neural network
I_a, I_c: The maximum number of iterations allowed for updating actor and critic networks in one time step
E_a, E_c: Thresholds to control whether an update can be terminated

$t = 0$
Actor network forward operation: compute $\mathbf{a}(t)$
Critic network forward operation: compute $\hat{J}[\mathbf{x}(t)]$
Output action $\mathbf{a}(t)$ and obtain the updated states $\mathbf{x}(t+1)$, reward $r[\mathbf{x}(t+1)]$, and termination request REQ_{term} from the environment or the plant
while $REQ_{term} \neq 1$ **do**
  $t = t+1$, $i_c = 0$, $i_a = 0$
  Actor network forward operation: compute $\mathbf{a}(t)$
  Critic network forward operation: compute $\hat{J}[\mathbf{x}(t)]$
  Compute the temporal difference $\delta(t)$
  while $\left(i_c < I_c \;\&\&\; \frac{\delta(t)^2}{2} \geq E_c\right)$ **do**
    Critic network backward operation: update $\mathbf{w^{c2}}$ and $\mathbf{w^{c1}}$
    Critic network forward operation: compute $\hat{J}[\mathbf{x}(t-1)]$
    Compute the temporal difference $\delta(t)$
    $i_c = i_c + 1$
  Compute the cost function $\frac{e_a^2}{2}$
  while $\left(i_a < I_a \;\&\&\; \frac{e_a^2}{2} \geq E_a\right)$ **do**
    Actor network backward operation: update $\mathbf{w^{a2}}$ and $\mathbf{w^{a1}}$
    Actor network forward operation: compute $\hat{J}[\mathbf{x}(t)]$
    Compute the cost function $\frac{e_a^2}{2}$
    $i_a = i_a + 1$
  Output action $\mathbf{a}(t)$ and obtain the updated states $\mathbf{x}(t+1)$, reward $r[\mathbf{x}(t+1)]$, and termination request REQ_{term} from the environment or plant

Output: $\mathbf{w^{a1}}$, $\mathbf{w^{a2}}$, $\mathbf{w^{c1}}$, $\mathbf{w^{c2}}$: updated weights for the actor and critic neural network

Fig. 2. Pseudocode for the ADHDP Algorithm.

multiplication operations. Similar to most machine-learning accelerators [23]–[25], [28], we adopt a tile-based matrix multiplication strategy, where the matrix is partitioned into several smaller blocks. The size of the tile is determined by the number of data lanes available in the system. In this design, the number of lanes is set to four, as this is enough for the applications targeted by this paper. Nevertheless, the proposed architecture and design methodology are scalable, so more lanes can readily be added into the design to accommodate larger problems.

For the forward operation, we adopt a row-wise multiplication. The neuron activation vector is first loaded from the neuron memory to the activation buffer. The activation buffer is a circular buffer, and it rotates a complete circle when multiplying each row in the matrix. Loading the data from neuron memory to the input buffer has the advantage that the data in the buffer can be reused without accessing the relatively-large neuron memory repeatedly, thereby saving power and time. Synaptic weights in the SRAM are arranged in a way such that weights corresponding to one tile are stored in the same row.

For the backward operation, there are two major steps: error backpropagation and weight update. The error backpropagation operation is also a matrix-vector multiplication. Similar to the forward operation, a tile-based multiplication is used. However, the multiplication in this case is done column-wise instead of row-wise. Such an arrangement has the advantage that the access for the memory is always sequential, providing a more regular memory access pattern when off-chip memory is used. In the weight update operation, two vectors are multiplied to form a matrix that is added to the old synaptic weight matrix. In this case, elements in the row vector are stored in the circular buffer, whereas elements associated with the column vector are stored in the linear buffer. The error backpropagation and weight update operations are scheduled in alternate clock cycles in order to reuse the same row of synaptic weights. Therefore, for one backward operation, each entry in the synaptic weight SRAM only needs to be read and written once.

B. Datapath

The datapath is partitioned into five/six-stage reconfigurable pipelines: schedule, fetch, multiply, add, activate, and write back.

1) Scheduling and Data-Fetching Operations: In the schedule stage, instructions fetched from the instruction memory are decoded to obtain the necessary information for scheduling operations with data. In the proposed single-instruction-multiple-data (SIMD) architecture, one instruction may contain the workloads that need multiple clock cycles to complete. Therefore, the instruction fetching and decoding occur selectively with the help of the controller. The scheduler needs to generate and latch the addresses for the data to be fetched in the fetch stage as well as to inspect any potential data hazard. Upon detecting that the data needed in the speculatively-scheduled operation are not ready in the input buffer, the scheduler looks for possibilities of data forwarding directly from the memory or write-back buffer. If even the data forwarding is not able to resolve the data hazard, a STALL operation is inserted into the pipeline as a null operation in order to wait for the data that are needed to be computed. In the fetch stage, data is read from the input buffers or memory and is latched in corresponding pipeline registers.

2) Arithematic and Write-Back Opeartions: The multiply and add stage conduct multiplication and addition operations, respectively. Adders in the add stage can be configured as parallel adders, or adder tree, or mixed of both depending on the operations conducted. The activate stage implements the activation function employed in neural networks. The hyperbolic tangent function is employed in this design, as it is the

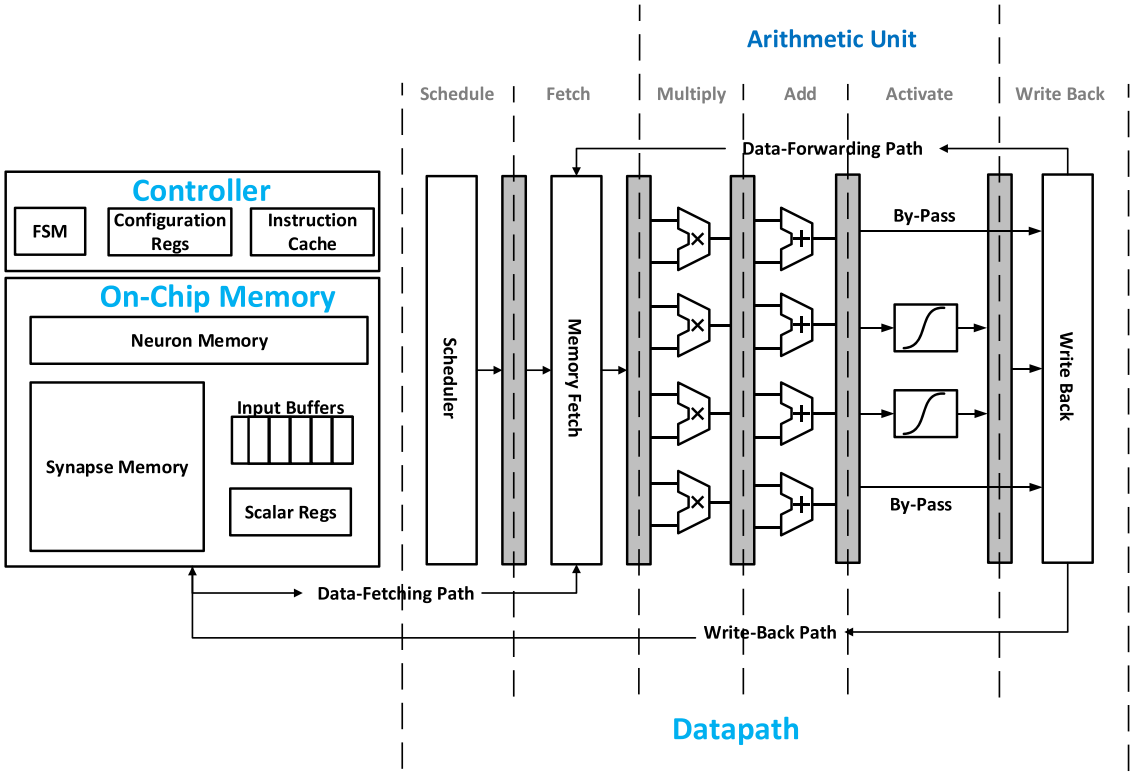

Fig. 3. Hardware architecture for the proposed accelerators. Data-level parallelism is exploited through utilizing multiple datapath lanes. A reconfigurable five/six-stage pipeline is used for the datapaths.

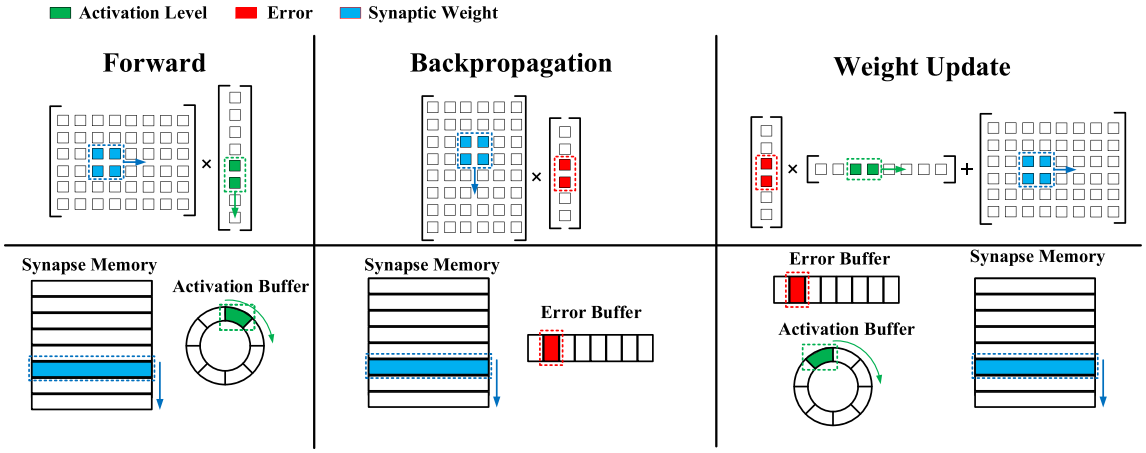

Fig. 4. Illustration of the data flow and memory access pattern in the proposed accelerators. Data buffers are employed to exploit the locality of the data. Synaptic weights needed in one tile-based operation is stored in one row.

most popular choice for ADP algorithm in the literature [6], [7], [13], [14], [16], [18]. In the proposed design, the activation function is implemented with piecewise linear interpolation, similar to those employed in [23] and [38]. Depending on the operation conducted, the activation stage may be bypassed, as the activation operation is only needed in the forward phase. In this case, the six-stage pipeline is reduced to a five-stage pipeline. After arithmetic computations, the write-back stage in the end of the pipe writes computed results back to storage units according to the instruction executed.

3) Datapath Quantization: One important consideration in designing customized accelerators is the choice of bitwidth used to represent data in the system. It is the norm to use a fixed-point number representation in machine-learning accelerators [23]–[36] because of its ease of implementation and good computational efficiency. To provide some guidelines in determining the proper bitwidth in our system, we conduct parametric simulations on the learning performance of the ADP algorithm under different bitwidths. Three most popular benchmark tasks for the ADP algorithm are employed in the

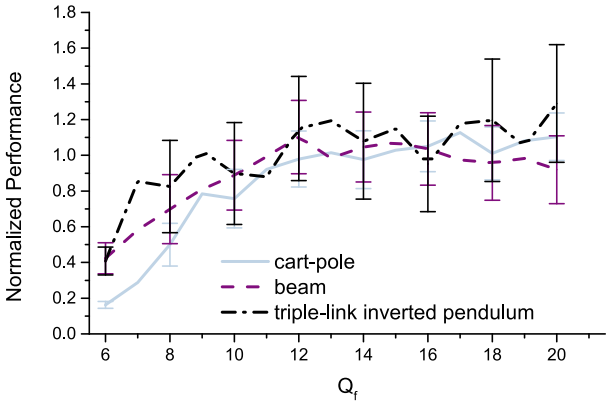

Fig. 5. Comparison of the learning performance achieved with different levels of data quantization for three classic ADP benchmarks. The obtained performances are normalized with respect to those obtained from the double-precision floating-point computations.

parametric study: The cart-pole balancing problem [6], [7], [13], [16], the beam-balancing problem [15], [17], and the triple-link inverted pendulum problem [6], [13]. In these three tasks, the target is to control the system such that the states of the system stay within some pre-defined ranges. In our experiment setting, once the states of the system under control exceed the desired ranges, a punishment (or a negative reward) is provided. For each task, data used in the algorithm, including synaptic weight, neuron states, and other intermediate variables, are quantized into numbers with a fractional bitwidth of Q_f. Learning processes are then conducted according to the ADP algorithm. The learning performance obtained for each task with different levels of quantizations are compared in Fig. 5. For each task, 50 runs are conducted, where a run contains several trials. Each trial is a complete process from beginning to end. A trial is ended either when the maximum time is reached (1000 time steps in these experiments), or the states under control exceed certain limits. The learning performance is measured by the total time that the plants are successfully maintained in the desired states in all trials. In Fig. 5, the performances obtained with different bitwidth are normalized to the performances obtained from computation with a double-precision floating-point number representation. The error bars in the figure correspond to the 95% confidence interval. As shown in the figure, performances achieved with quantized data start matching those obtained with double-precision data when the bitwidth for the fractional part reaches 12 bit. We use a 6-bit integer part (including a 1-bit sign information) and an 18-bit fractional part to represent data in our accelerators. The extra 6 bits in the fractional part compared to the 12-bit lower limit is to provide some tolerance in the design.

C. Controller

The main role of the controller is to determine the instruction flow. The format of instructions developed for our accelerator is shown in Fig. 6(a). The operation code field specifies the type of instruction. There are six types of instructions in our accelerator, as shown in Fig. 6(b). The code "FF" corresponds to the forward operation, which is the most common instruction. The code "SCA" is for scalar operation such as calculating the temporal difference as shown in (4). The operation code "BP_WU" is used for hidden-layer units where both error backpropagation and weight update are needed. Code "BP" and "WU" are for error backpropagation and weight update, respectively. They are used when only one operation is needed. For example, "WU" code can be used for the input layer when error backpropagations are not needed. The code "CC" calls for controller operation. It can be used, for example, to implement the conditional jumps in Fig. 2. The fields "Source Addr", "Synapse Addr", and "Destination Addr" specify the addresses for the source data, the address for the synaptic weight and the addresses to write back, respectively. In our design, the synapse memory has its own address space, whereas all other storage units share a unified address space. The "# of Row" and "# of Column" fields indicate the size of the matrix. The filed "Offset" specifies any offset in computing the matrix multiplication. For example, as shown in Fig. 2, backpropagation for the actor network only needs to be done for $\mathbf{a}(t)$. Therefore, elements associated with $\mathbf{x}(t)$ should be skipped through specifying the offset. The "Config" field is used for configuration purposes, for example, to specify whether to bypass the activation stage in the datapath.

With all the fields specified in the instructions, the scheduler can schedule operations based on this information. In the proposed architecture, one instruction specifies all operations conducted on one matrix. An example of the instructions corresponding to the pseudocode shown in Fig. 2 is illustrated in Fig. 6(c). Only a portion of the instructions are shown for the purpose of brevity. All instructions in the figure correspond to a series of operations conducted by the datapath except for the "Check Point" operation where the controller conducts a conditional jump with the help of an FSM.

IV. Virtual Update Technique

In this section, we examine a few unique features of the ADP algorithm. These features are then exploited to improve the speed and energy efficiency of the accelerator.

A. Algorithm

In the ADP algorithm, it is the norm to conduct many internal cycles in order to minimize the cost function for each input vector [6]–[19]. This corresponds to the second or the third *while* loop in Fig. 2. The maximum number of internal loops for each input vector is typically in the range of 10 to 100. In other words, many iterations are carried out for the same input vector, attempting to minimize the cost function at the current time step. Therefore, it may be worth conducting some pre-processing if the same input vector is used repeatedly. Such a simplification is indeed possible by inspecting the unrolled *while* loop.

Without loss of generality, let us focus on the update in the critic network. Weights in the input layer of the critic network are updated according to (11), followed by the immediate forward operation in (8). For the ease of explanation, (11) and (8)

Op Code	Source Addr	Synapase Addr	Destination Addr	# of Row	# of Column	Offset	Config

(a)

Op Code	Operation
FF	Forward Operation
SCA	Scalar Operation
BP_WU	Error Backpropagation & Weight Update
BP	Error Backpropagation
WU	Weight Update
CC	Controller Operation

(b)

```
0  Check Point
1  Actor Feedforward          -> FF 0 0 0 3 2 0 0
                                 FF 2 6 8 1 3 0 0
                                 FF 6 9 10 3 3 0 0
                                 FF 10 18 128 1 3 0 1
5  Check Point
6  Actor Feedforward
10 Check Point
11 Critic Feedforward
13 Scalar Operation
14 Check Point
15 Critic Backpropagation     -> BP_WU 10 18 0 1 3 0 0
                                 BP 6 9 0 3 3 0 0
17 Check Point
18 Actor Feedforward
22 Scalar Operation
23 Check Point                -> CC 0 0 0 0 0 0 0
24 Actor Backpropagation
28 Check Point
```

(c)

Fig. 6. Illustration of the instructions used in the accelerators. (a) Format of the instruction. (b) List of all operation codes and their corresponding operations. (c) A sample program for implementing the ADHDP algorithm shown in Fig. 2.

are rewritten in (14) and (15) with the dependence on the loop index i_c explicitly indicated. Note that $\mathbf{p^c}$ in the equations is not a function of i_c, as the state vector and selected action remain the same when the critic network is updating.

$$\Delta w_{ij}^{c1}(i_c) = \alpha e_i^{c1}(i_c)\sigma'\left[\sum_{k=1}^{m+n} w_{ik}^{c1}(i_c)p_k^c\right]p_j^c \tag{14}$$

$$h_i^c(i_c+1) = \sigma\left[\sum_{j=1}^{m+n} w_{ij}^{c1}(i_c+1)p_j^c\right] \tag{15}$$

By substituting (15) into (14) with the help of the relationship $w_{ij}^{c1}(i_c+1) = w_{ij}^{c1}(i_c) + \Delta w_{ij}^{c1}(i_c)$, one can obtain

$$h_i^c(i_c+1) = \sigma\left[o_i^c(i_c+1)\right] \tag{16}$$

where

$$o_i^c(i_c+1) = o_i^c(i_c) + \epsilon_i(i_c)\Lambda_c \tag{17}$$

$$o_i^c(i_c) = \sum_{j=1}^{m+n} w_{ij}^{c1}(i_c)p_j^c \tag{18}$$

$$\epsilon_i(i_c) = \alpha e_i^{c1}(i_c)\sigma'\left[o_i^c(i_c)\right] \tag{19}$$

$$\Lambda_c = \sum_{j=1}^{m+n}(p_j^c)^2 \tag{20}$$

$o_i^c(i_c)$ is the input to neuron h_i^c in the i_c^{th} iteration; $\epsilon_i(i_c)$ is the scaled backpropagated error at the input of neuron h_i^c in the i_c^{th} iteration. These two quantities are computed in the i_c^{th} iteration for the forward and backward operations, respectively. Λ_c is the sum of squares of activation levels of all input neurons, which is independent of i_c. Therefore, activation levels of the hidden layer units in the $(i_c+1)^{th}$ iteration can be conveniently calculated based on results obtained from the i_c^{th} iteration. Even though no actual weight update or forward operations are conducted, it appears to neurons in other layers as if weights in the input layer were updated. We call this technique the virtual update technique.

More conveniently, if the virtual update technique is employed from the 0^{th} iteration, then we have

$$o_i^c(i_c) = o_i^c(0) + E_i(i_c)\Lambda_c \tag{21}$$

$$o_i^c(0) = \sum_{j=1}^{m+n} w_{ij}^{c1}(0)p_j^c \tag{22}$$

$$E_i(i_c) = \sum_{k=0}^{i_c} \alpha e_i^{c1}(k)\sigma'\left[o_i^c(k)\right] \tag{23}$$

In this case, we only need to update $E_i(i_c)$ in each iteration, whereas $o_i^c(0)$ and Λ_c remain the same. When the update loop is terminated either because the maximum number of iterations I_c is reached or because the cost function is below a certain threshold, synaptic weights associated with the input layer are updated according to (24).

$$\Delta w_{ij}^{c1} = E_i(i_c)p_j^c \tag{24}$$

TABLE I
COMPARISON OF COMPUTATIONAL COMPLEXITY OF CONVENTIONAL UPDATE AND VIRTUAL UPDATE

	Regular Update	Virtual Update
Forward	$N_i N_h L$ **MAC**	$(L-1)N_h$ **MAC** $+N_i$ **MUL** $+N_i N_h$ **MAC**
Backward	$N_i N_h L$ **MAC**	$(L-1)N_h$ **ADD** $+N_i N_h$ **MAC**
Total Operations (**MAC/MUL/ADD**)	$2N_i N_h L$	$2(L+N_i-1)N_h+N_i$
Complexity	$O(N_i N_h L)$	$O[(L+N_i)N_h]$

It is worth mentioning that the proposed virtual update algorithm does not reduce the precision of the algorithm. That is, the speedup of the algorithm is achieved through reordering effective operations more efficiently instead of using approximations.

B. Implementation Considerations

In order to exploit the proposed virtual update technique, one extra instruction "VU" is added to our instruction set introduced in Section III. This instruction implements (21) - (23) in two groups of operations. The first group of operations is to compute and store Λ_c when the current input vector is presented for the first time. The second group of operations are the multiply-and-add operations shown in (21). The operation of accumulating $E_i(i_c)$ is merged to the normal "BP" or "BP_WU" operations without introducing any overhead in computational time. It is worth noting that all newly-added operations, as shown in (21)-(23), scale linearly with the size of the network, whereas the original backward and forward operations scale quadratically. Therefore, the virtual update technique can help save significant computational efforts.

To implement the virtual update algorithm, $o_i^c(0)$ and $E_i(i_c)$ need to be stored. They can be stored conveniently in the synapse memory, recognizing that the weight memory is not utilized during the virtual update operation. Indeed, the virtual update technique avoids both writing synaptic weights in the weight update phase and reading weights in the forward phase, leaving the weight memory free during that period. Compared to synaptic weights stored in the synapse SRAM, the additional memory overhead caused by the virtual update technique is negligible, especially when the size of the network is large.

The pseudocode for the *while* loop of updating the critic network with the proposed virtual update technique is shown in Fig. 7. If the current iteration is not the last iteration allowed by the maximum number of iterations, the virtual update algorithm is used to compute neuron activation levels of hidden-layer neurons in the next iteration, otherwise the conventional update is used. It should be ensured that when exiting the while loop, normal weight update has to be conducted once according to (24), as weights are actually not updated during previous iterations.

The computational complexity of the virtual update algorithm is compared with that of the baseline in Table I. The number of arithmetic operations per time step is used for comparison. In the table, N_i and N_h represent the number of input-layer neurons and hidden-layer neurons, respectively. L specifies the number of iterations in one time step. Its value should be in the range of $[1, I_c]$ or $[1, I_a]$ depending on whether the network is critic or actor. "MAC", "MUL" and "ADD" denote multiply-accumulate, multiply, and add operations, respectively. It is worth noting that complexity listed in the table is valid for the case where $L > 1$. When $L = 1$, two algorithms have the same computational complexity, as no virtual update takes place. As shown in the table, the virtual update technique significantly reduces the number of operations needed. Even though the technique is only applicable for synapses between input layer and hidden layer, the savings in computational efforts is remarkable, as most weights in neural networks concentrate in between these two layers. The actual percentage of savings is reported in Section V.

```
while (i_c < I_c && δ(t)^2/2 ≥ E_c) do
    Backward operation: update w^c2
    if (i_c == I_c − 1) then
        Backward operation: update w^c1
        Forward operation: compute h^c
    else
        Virtual update: compute h^c
    Forward operation: compute Ĵ[x(t − 1)]
    Compute the temporal difference δ(t)
    if (δ(t)^2/2 < E_c) then
        Backward operation: update w^c1
    i_c = i_c + 1
```

Fig. 7. Pseudocode for the *while* loop corresponding to the critic update when the virtual update algorithm is employed.

V. DESIGN EXAMPLES

Hardware architectures and techniques discussed in previous sections are implemented in TSMC 65-nm CMOS technology, and the obtained simulation results are presented in this section. In order to examine all aspects of the proposed design methodology and design strategies, simulators are developed in a high-level programming language. These simulators model behaviors of the final chip, and they are employed to measure the input-output relationships and clock cycle needed to accomplish certain tasks. Area, speed and power consumption are evaluated based on post-layout circuit simulation results.

Two accelerators are implemented. One implementation is equiped with the proposed virtual update algorithm, whereas another one is the baseline design with the conventional update. Both chips have similar chip layouts. Therefore, only the one with virtual update is shown in Fig. 8 for brevity.

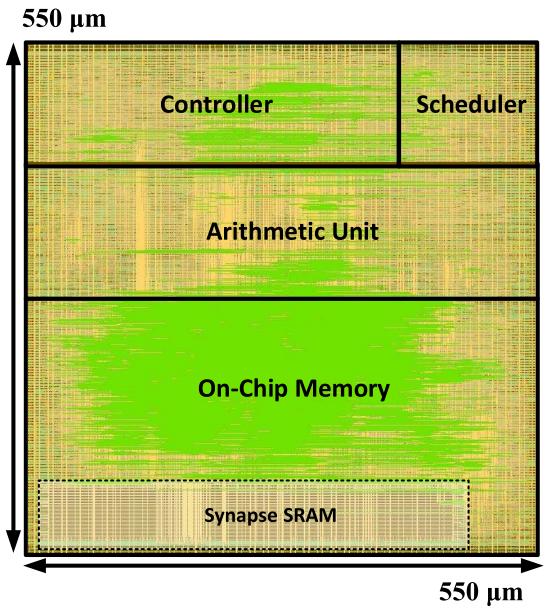

Fig. 8. Chip layout and floorplan of the accelerator chip with the virtual update algorithm.

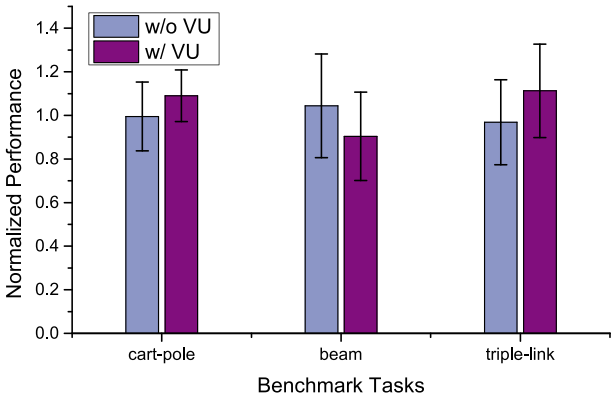

Fig. 9. Comparison of the learning performances achieved by the accelerators and the software approach for three commonly used benchmarks. The results obtained from the accelerators are normalized to those obtained from software. Error bars correspond to a confidence interval of 95%.

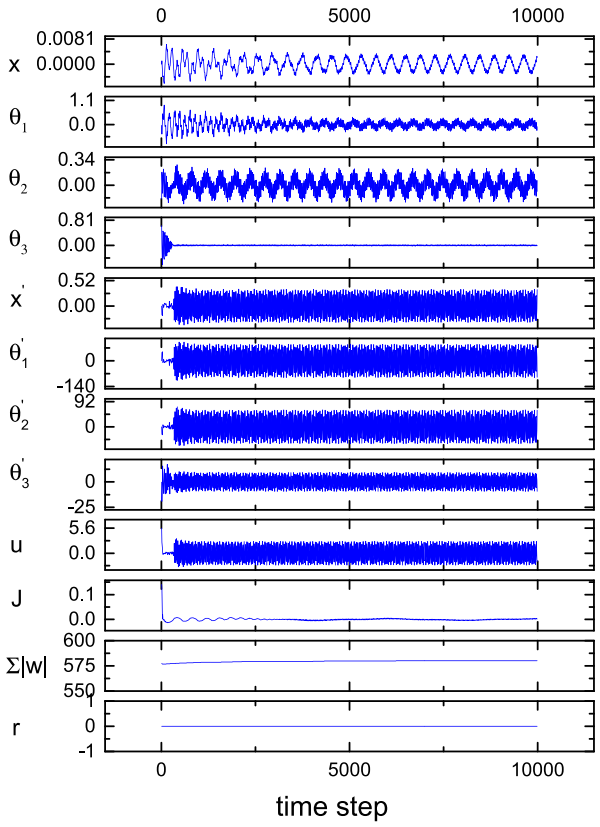

Fig. 10. Typical waveforms obtained in the triple-link inverted pendulum task with the baseline accelerator. In the figure, the unit for distances and angles are meter and degree, respectively.

On-chip memories, including synaptic weight memory, neuron memory, input buffers, and instruction memory take most of the spaces. The arithmetic unit, which contains multipliers, adders, and the activation block, is the second largest block. The controller and scheduler occupy the rest of the area. In this design, synaptic weights are stored in a 1536-byte SRAM array. The sizes of the registers for storing neuron activations, intermediate results, and instructions are 192 bytes, 336 bytes and 240 bytes, respectively. Both the baseline and the upgraded accelerators are designed to operate at a clock frequency of 175 MHz. Such a clock frequency is more than enough for allowing all benchmark tasks discussed in this paper to operate in real time.

In order to evaluate the performance of the accelerators in conducting reinforcement-learning tasks, three most common control benchmarks used in Section III are employed. The same metric, accumulated time steps, is used for comparison. The performance achieved by the accelerator is normalized with respect to the performance achieved by the software approach implemented on a general-purpose processor. The obtained results are shown in Fig. 9. Since the virtual update algorithm does not use any approximation or assumption in the computation, the obtained results should be the same as the baseline design when quantization is absent. Nevertheless, there exists slight differences in the computed results caused by different ordering of quantizations. The results in the figure are obtained from the behavior-level model of the chip. Mathematical models used for simulating these benchmark tasks can be found in [6], [7], [13], and [15]–[17], and they are omitted in this paper for brevity. As shown in the figure, the accelerators are able to achieve a similar performance compared to the processor that computes with double-precision floating-point numbers. One set of typical waveforms obtained from the triple-link task for a successful learning is demonstrated in Fig. 10. In the figure, x, θ_1 - θ_3 and their corresponding derivative x' and θ_1' - θ_3' are eight state variables under control, u is the applied control voltage, J is the estimated reward-to-go, $\Sigma|\omega|$ is the sum of absolute values of all weights, and r is the reward signal, which is -1 if the states of the plant exceed the target range. Initial conditions for the plants are set as the following. x and x' are initialized as zero. $\theta_1 - \theta_3$ and $\theta_1' - \theta_3'$ are initialized randomly. They obey uniform distributions $U\,[-1°, 1°]$ and $U\,[-0.5°/s, 0.5°/s]$, respectively. The target is to control x to

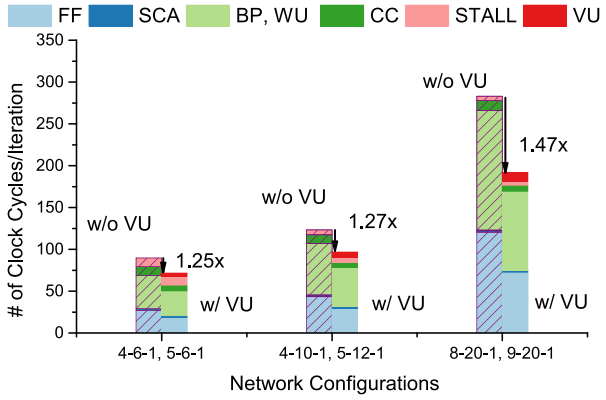

Fig. 11. Comparison of the numbers of clock cycles needed for every critic/actor update iteration. The first two groups of data are obtained from the cart-pole balancing task, whereas the third group of data is from the triple-link inverted pendulum task. The forward and backward operations consume most clock cycles. The overheads of scalar operation and control operation are quickly diluted as the sizes of the neural networks increase.

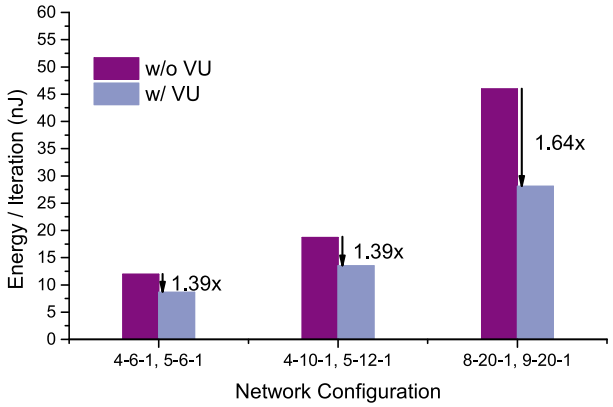

Fig. 13. Comparison of the energy consumption for every critic/actor update iteration. The first two groups of data are obtained from the cart-pole balancing task, whereas the third group of data is from the triple-link inverted pendulum task. The virtual update technique effectively improves the energy efficiency. The improvement is more significant for larger neural networks.

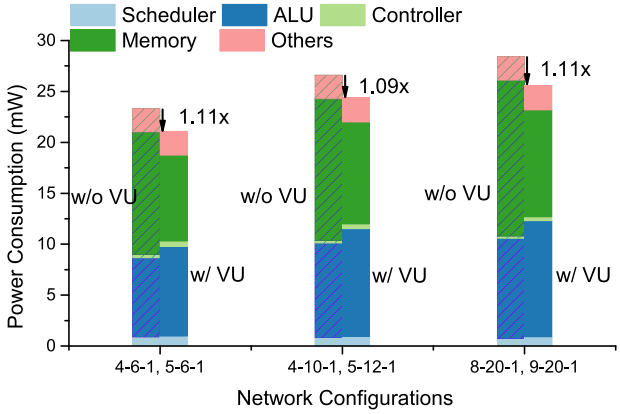

Fig. 12. Comparison of the power consumption breakdown for every critic/actor update iteration. The first two groups of data are obtained from the cart-pole balancing task, whereas the third group of data is from the triple-link inverted pendulum task. The arithmetic unit and memory consume most of the power. The virtual update technique improves the power consumption slightly.

be within the range of $[-1\text{m}, 1\text{m}]$ and to maintain θ_1 - θ_3 in the range of $[-20°, 20°]$, while the applied voltage is bounded by $\pm 30V$. More detailed information on the triple-link inverted pendulum balancing task can be found in [6] and [13]. It is demonstrated in Fig. 10 that the accelerator successfully learns the control policy and maintains the states of the system well within the target range.

To demonstrate the effectiveness of the proposed virtual update technique, Fig. 11-Fig. 13 compare the number of clock cycles, power consumption, and energy efficiency of the accelerators with and without the proposed technique. Different sizes of neural networks and control tasks are examined. In the figures, results with labels "4-6-1, 5-6-1" and "4-10-1, 5-12-1" are obtained from the cart-pole task, where the two sets of numbers indicate the sizes of the critic and actor networks, respectively. The results with the label "8-20-1, 9-20-1" are obtained from the triple-link inverted pendulum task. In addition, specifications of the proposed ADHDP accelerator are summarized in Table II. The ADP accelerator faces many unique challenges compared to other machine-learning accelerators, as pointed out in Section I. Nevertheless, to provide some insights, two state-of-the-art accelerators for supervised learning are also presented in the table. Two trends can be observed in Table II. The ADP accelerator is smaller compared to the accelerators for supervised-learning tasks. Indeed, most ADP algorithms deal with extracted features, which can be handled by networks with smaller sizes. Nevertheless, the proposed design methodologies are scalable, and the accelerator can be readily scaled to deal with larger problems. The second observation is that the number of bit used in the ADP accelerator is larger than those used in supervised-learning accelerators. This is mainly because learning typically requires a higher numerical precision compared to inference in order to converge.

The normalized number of clock cycle breakdowns are compared in Fig. 11. One trend that can be observed for both the baseline accelerator and the accelerator with the virtual update is that the forward and backward operations occupy most of the clock cycles, and the percentages that these two operations occupy increase as the size of neural networks becomes large. Indeed, as the size of the neural network increases, the number of operations that can flow through the pipeline without being interrupted by the control or branch operation increases. The proposed virtual update algorithm effectively shortens the number of clock cycles needed for each task. The improvement grows as the size of the problem increases. A 1.47 times improvement is achieved for the triple-link inverted pendulum benchmark task. The main reason for the growing improvement is that the virtual update algorithm effectively replaces the quadratically-scaled operations with linearly-scaled operations. Therefore, the savings in the number of clock cycles increases with the size of the problem. To give a comparison between the accelerators presented in this paper and a software running on a general-purpose processor, the ADP algorithm is programmed and is run on an Intel Xeon processor. The improvement in running

TABLE II
SUMMARY AND COMPARISON OF SPECIFICATIONS OF THE ADHDP ACCELERATOR

	This work	[25]	[23]
Application	Optimal Control	Classification	Classfication
Algorithm	ADP/reinforcement learning	supervised learning	supervised learning
On-Chip/Real-Time Learning	Yes	No	No
Technology	TSMC 65nm	TSMC 65 nm	65 nm
Area	550 μm $\times$ 550 μm	3.5 mm $\times$ 3.5 mm	3 mm^2
Number of Lanes	4	168	256
Arithmetic Precision	24-bit fixed-point	16-bit fixed-point	16-bit fixed-point
Supply Voltage	1.2 V	1 V	-
Clock Frequency	175MHz	200 MHz	980 MHz
Power Consumption	25 mW	278 mW	485 mW

time which is averaged over all three network sizes is about 270 times.

Fig. 12 compares the power consumption of the two accelerators. The accelerator with virtual update has a slightly lower power consumption compared to the baseline design, thanks to many fewer memory operations, as illustrated in the figure. Another observation made from Fig. 12 is that the virtual update tends to increase the power consumption in the arithmetic unit. This can be attributed to two reasons. The first reason is that the virtual update increases the utilization rate of the arithmetic unit. Another reason is that multiplexers are added in the arithmetic unit to allow more operations needed in the virtual update algorithm, which contributes to the additional power. The additional power in the arithmetic unit is offset by the savings in memory operations, resulting in a net savings in the power consumption. The energy efficiencies are compared in Fig. 13. Through accumulating the improvements in both the number of clock cycles per iteration and the power consumption, energy efficiency of the accelerator with the virtual update technique has been improved as many as 1.64 times for the triple-link inverted pendulum task. Again, as the sizes of the network increase, the improvement in energy efficiency grows.

VI. CONCLUSION

In this paper, we present a hardware architecture for ADHDP accelerators. Through leveraging the data-level parallelism and data locality, scalable and programmable accelerators with high throughput and high energy efficiency are demonstrated. In addition, to exploit the iterative nature of the ADP algorithm, a virtual update technique is proposed to skip unnecessary computations, improving the throughput and power consumption. We demonstrate two design examples that are with and without the proposed technique. Extensive simulations are conducted to demonstrate efficacy of the design strategies and techniques. It is observed from the simulations that the accelerator is around 270 times faster than the software approach running on a general-purpose processor while achieving similar learning performance. Furthermore, the proposed virtual update algorithm can effectively improve the energy efficiency of the accelerator by a factor of 1.64 for the most complicated benchmark task we employ. Such a good energy efficiency and high throughput open the door for complicated ADP algorithms to be deployed in various energy-constraint applications where optimal decision-making or control are needed.

REFERENCES

[1] D. V. Prokhorov and D. C. Wunsch, "Adaptive critic designs," *IEEE Trans. Neural Netw.*, vol. 8, no. 5, pp. 997–1007, Sep. 1997.

[2] F. L. Lewis and D. Vrabie, "Reinforcement learning and adaptive dynamic programming for feedback control," *IEEE Circuits Syst. Mag.*, vol. 9, no. 3, pp. 32–50, 3rd Quart., 2009.

[3] F.-Y. Wang, H. Zhang, and D. Liu, "Adaptive dynamic programming: An introduction," *IEEE Comput. Intell. Mag.*, vol. 4, no. 2, pp. 39–47, May 2009.

[4] F. L. Lewis, D. Vrabie, and K. G. Vamvoudakis, "Reinforcement learning and feedback control: Using natural decision methods to design optimal adaptive controllers," *IEEE Control Syst.*, vol. 32, no. 6, pp. 76–105, Dec. 2012.

[5] D. Wang, H. He, and D. Liu, "Adaptive critic nonlinear robust control: A survey," *IEEE Trans. Cybern.*, vol. 47, no. 10, pp. 3429–3451, Oct. 2017.

[6] J. Si and Y.-T. Wang, "Online learning control by association and reinforcement," *IEEE Trans. Neural Netw.*, vol. 12, no. 2, pp. 264–276, Mar. 2001.

[7] D. Liu, X. Xiong, and Y. Zhang, "Action-dependent adaptive critic designs," in *Proc. IEEE Int. Joint Conf. Neural Netw. (IJCNN)*, vol. 2. Jul. 2001, pp. 990–995.

[8] M. S. Iyer and D. C. Wunsch, "Dynamic re-optimization of a fed-batch fermentor using adaptive critic designs," *IEEE Trans. Neural Netw.*, vol. 12, no. 6, pp. 1433–1444, Nov. 2001.

[9] D. Han and S. N. Balakrishnan, "State-constrained agile missile control with adaptive-critic-based neural networks," *IEEE Trans. Control Syst. Technol.*, vol. 10, no. 4, pp. 481–489, Jul. 2002.

[10] S. Ferrari and R. F. Stengel, "Online adaptive critic flight control," *J. Guid., Control, Dyn.*, vol. 27, no. 5, pp. 777–786, 2004.

[11] C.-K. Lin, "Adaptive critic autopilot design of Bank-to-turn missiles using fuzzy basis function networks," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 35, no. 2, pp. 197–207, Apr. 2005.

[12] S. Ferrari, J. E. Steck, and R. Chandramohan, "Adaptive feedback control by constrained approximate dynamic programming," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 38, no. 4, pp. 982–987, Aug. 2008.

[13] H. He, Z. Ni, and J. Fu, "A three-network architecture for on-line learning and optimization based on adaptive dynamic programming," *Neurocomputing*, vol. 78, no. 1, pp. 3–13, 2012.

[14] F. Liu, J. Sun, J. Si, W. Guo, and S. Mei, "A boundedness result for the direct heuristic dynamic programming," *Neural Netw.*, vol. 32, pp. 229–235, Aug. 2012.

[15] Z. Ni, H. He, and J. Wen, "Adaptive learning in tracking control based on the dual critic network design," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 24, no. 6, pp. 913–928, Jun. 2013.

[16] Y. Sokolov, R. Kozma, L. D. Werbos, and P. J. Werbos, "Complete stability analysis of a heuristic approximate dynamic programming control design," *Automatica*, vol. 59, pp. 9–18, Sep. 2015.

[17] Z. Ni, H. He, X. Zhong, and D. V. Prokhorov, "Model-free dual heuristic dynamic programming," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 26, no. 8, pp. 1834–1839, Aug. 2013.

[18] C. Mu, Z. Ni, C. Sun, and H. He, "Air-breathing hypersonic vehicle tracking control based on adaptive dynamic programming," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 3, pp. 584–598, Mar. 2017.

[19] Q. Wei, D. Liu, F. L. Lewis, Y. Liu, and J. Zhang, "Mixed iterative adaptive dynamic programming for optimal battery energy control in smart residential microgrids," *IEEE Trans. Ind. Electron.*, vol. 64, no. 5, pp. 4110–4120, May 2017.
[20] R. J. Wood, "The first takeoff of a biologically inspired at-scale robotic insect," *IEEE Trans. Robot.*, vol. 24, no. 2, pp. 341–347, Apr. 2008.
[21] N. O. Pérez-Arancibia, K. Y. Ma, K. C. Galloway, J. D. Greenberg, and R. J. Wood, "First controlled vertical flight of a biologically inspired microrobot," *Bioinspiration Biomimetics*, vol. 6, no. 3, p. 036009, 2011.
[22] P. Mazumder, D. Hu, I. Ebong, X. Zhang, Z. Xu, and S. Ferrari, "Digital implementation of a virtual insect trained by spike-timing dependent plasticity," *Integr., VLSI J.*, vol. 54, pp. 109–117, Jun. 2016.
[23] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, 2014.
[24] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp. 609–622.
[25] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[26] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. 43rd Int. Symp. Comput. Archit.*, Jun. 2016, pp. 267–278.
[27] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Seoul, South Korea, 2016, pp. 1–13.
[28] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Trans. Comput.-Aided Des. Integr.*, vol. 36, no. 3, pp. 513–517, Mar. 2017.
[29] Y. Sun and A. C. Cheng, "Machine learning on-a-chip: A high-performance low-power reusable neuron architecture for artificial neural networks in ecg classifications," *Comput. Biol. Med.*, vol. 42, no. 7, pp. 751–757, 2012.
[30] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *Proc. 39th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2012, pp. 356–367.
[31] J. Kung, D. Kim, and S. Mukhopadhyay, "A power-aware digital feed-forward neural network platform with backpropagation driven approximate synapses," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Des. (ISLPED)*, Jul. 2015, pp. 85–90.
[32] A. Savich, M. Moussa, and S. Areibi, "A scalable pipelined architecture for real-time computation of MLP-BP neural networks," *Microprocess. Microsyst.*, vol. 36, no. 2, pp. 138–150, 2012.
[33] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 27–40.
[34] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
[35] S. Venkataramani *et al.*, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 13–26.
[36] N. Zheng and P. Mazumder, "A low-power circuit for adaptive dynamic programming," in *Proc. 31th Int. Conf. VLSI Des. (VLSID)*, Jan. 2018.
[37] C. Bishop, *Pattern Recognition and Machine Learning* (Information Science and Statistics). Secaucus, NJ, USA: Springer-Verlag, 2006.
[38] D. Larkin, A. Kinane, V. Muresan, and N. O'Connor, "An efficient hardware architecture for a neural network activation function generator," in *Proc. Int. Symp. Neural Netw.*, 2006, pp. 1319–1327.

Nan Zheng (S'13) received the B.S. degree in information engineering from Shanghai Jiao Tong University, Shanghai, China, in 2011, and the M.S. degree in electrical engineering from the University of Michigan, Ann Arbor, in 2014, where he is currently pursuing the Ph.D. degree in electrical engineering.

In 2012, he had an internship at Qualcomm, CA, where he involved in developing antenna system for the next-generation communication network. His research interests include low-power circuit design, modeling, and optimization with an emphasis on machine-learning applications.

Pinaki Mazumder (S'84–M'87–SM'95–F'99) received the Ph.D. degree from the University of Illinois at Urbana–Champaign, Urbana, in 1988.

He is currently a Professor with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. He was with industrial research and development centers that included AT&T Bell Laboratories for six years, where in 1985, he started the CONES Project the first C modeling-based very large scale integration (VLSI) synthesis tool at Indias premier electronics company, Bharat Electronics Ltd., India, where he had developed several high-speed and high-voltage analog integrated circuits intended for consumer electronics products. He has authored or co-authored over 320 technical papers and five books on various aspects of VLSI research works. His current research interests include current problems in nanoscale CMOS VLSI design, computer-aided design tools, and circuit designs for emerging technologies, including quantum MOS and resonant tunneling devices, semiconductor memory systems, and physical synthesis of VLSI chips.

Dr. Mazumder is a fellow of the American Association for the Advancement of Science in 2008. He was a recipient of the Digitals Incentives for Excellence Award, the BF Goodrich National Collegiate Invention Award, and the Defense Advanced Research Projects Agency Research Excellence Award.