Online Autonomous Layered Learning in Dynamic Multimedia Systems

Nicholas Mastronarde*, Mihaela van der Schaar

ABSTRACT

In our previous paper, we proposed a systematic cross-layer framework for dynamic multimedia systems, which allows each layer (i.e. application, operating system, and hardware) to make *autonomous* and *foresighted* decisions that maximize the system's *long-term performance*. The proposed solution solves the cross-layer optimization *offline*, under the assumption that the multimedia system's probabilistic dynamics (e.g. the application's rate-distortion-complexity behavior) are *known* a priori, by modeling the system as a layered Markov decision process (MDP). In practice, however, these dynamics are *unknown* a priori and therefore must be learned *online*, at run-time. In this paper, we extend our previous paper to address this problem. In particular, we allow the multimedia system layers to learn, through repeated interactions with each other, to autonomously optimize the system's long-term performance at run-time. We propose several novel decentralized (layered) reinforcement learning algorithms and analyze their required computation, memory, and communication overheads. In our experiments, we demonstrate that decentralized learning can perform as well as centralized learning.

I. INTRODUCTION

State-of-the-art multimedia technology is poised to enable widespread proliferation of a variety of life-enhancing applications, such as video conferencing, emergency services, surveillance, telemedicine, remote teaching and training, augmented reality, and distributed gaming. However, efficiently designing and implementing such delay-sensitive multimedia applications on resource-constrained, heterogeneous devices and systems is challenging due to the real-time constraints, high workload complexity, and the time-varying environmental dynamics experienced by the system (e.g. video source characteristics, user requirements, workload characteristics, number of running applications, memory/cache behavior etc.).

Cross-layer adaptation is an increasingly popular solution for addressing these challenges in

dynamic multimedia systems (DMSs) [1]-[8]. This is because the performance of DMSs (e.g. video rate-distortion costs, delay, and power consumption) can be significantly improved by jointly optimizing parameters, configurations, and algorithms across two or more system layers (i.e. the application, operating system, and hardware layers), rather than optimizing and adapting them in isolation. However, existing cross-layer solutions have several important limitations.

Centralized cross-layer solutions: The majority of these solutions require a central optimizer to coordinate the application, operating system, and hardware adaptations to optimize the performance of one or multiple multimedia applications sharing the DMS's limited resources. To this end, a new interface between the central optimizer and all of the layers is created, requiring extensive modifications to the operating system [4] [5] or the introduction of an entirely new middleware layer [1] [2] [3] [24] [25]. However, these approaches violate the layered system architecture because individual layers no longer have control of their actions; and, they complicate the system's design and increase implementation costs, because individual layers cannot be upgraded without requiring a significant system redesign [8] [18] [19].

Myopic cross-layer solutions: Another limitation of many existing cross-layer solutions for DMSs is that they are *myopic*. In other words, cross-layer decisions are made reactively in order to optimize the *immediate* utility, without considering the impact of these decisions on the future utility. However, in DMSs, it is essential to predict the impact of the current decisions on the long-term utility because the multimedia source characteristics, workload characteristics, OS and hardware dynamics are often correlated across time. Moreover, in DMSs, utility fluctuations across time lead to poor user experience and bad resource planning leads to inefficient resource usage and wasted power [26] [27]. In contrast, *foresighted* (i.e. long-term) optimization techniques take into account the impact of immediate cross-layer decisions on the DMS's expected future utility. Importantly, foresighted policy optimizations have been successfully deployed at the hardware layer to solve the dynamic power management problem [11]-[13]; however, with the exception of our prior paper [9], they have never been used for cross-layer DMS optimization, where *all the layers make foresighted decisions*.

Single-layer learning solutions: The various multimedia system layers must be able to adapt at run-time to their experienced dynamics. Most existing learning solutions for multimedia systems are concerned with modeling the unknown and potentially time-varying environment experienced by a single layer. We refer to these as *single-agent* learning solutions (in our setting,

an agent corresponds to a layer). A broad range of single-agent techniques have been deployed in multimedia (and general purpose) systems in recent years, which include estimation techniques such as maximum likelihood estimation (MLE) [4] [5] [10] [12] [13], statistical fitting [7], regression methods [28], adaptive linear prediction [29], and ad-hoc estimation heuristics [3]. However, these solutions ignore the fact that DMSs do not only experience dynamics at a single layer (e.g. time-varying workload), but at multiple (possibly all) layers, which interact with each other. Hence, it is necessary to enable the layers to learn at run-time how to autonomously and asynchronously adapt their processing strategies based on forecasts about (i) their *future dynamics* and, importantly, (ii) *how they impact and are impacted by the other layers*. For this reason, we will model the run-time cross-layer optimization as a cooperative multi-agent learning problem [30] (where the layers are the agents). The key challenge in this multi-agent learning setting is that each layer's learning performance is directly impacted by not only the environment dynamics, but also by the learning processes of the other layers with which it interacts.

In summary, while significant contributions have been made to enhance the performance of DMSs using cross-layer design techniques, no systematic cross-layer framework exists that explicitly considers: (i) the design and information constraints imposed by a layered DMS architecture; (ii) the ability of the various layers to autonomously make foresighted decisions in order to jointly maximize the DMS's utility; and, most importantly, (iii) the need for layers to learn on-line their unknown environmental dynamics and how they impact and are impacted by the other layers.

In this paper, similar to [6] [7] [8], we consider a multimedia system in which (i) a video encoder at the application layer makes rate-distortion-complexity tradeoffs by adapting its configuration and (ii) the operating system layer makes energy-delay tradeoffs by adapting the hardware layer's operating frequency. Our contributions are as follows:

- We propose two novel multi-agent (layered) reinforcement learning algorithms, which allow the multimedia system layers to autonomously learn their optimal foresighted policies online, through repeated interactions with each other. We show experimentally that one of the proposed algorithms, which adheres to the layered system architecture, performs as well as a traditional centralized learning algorithm, which does not.
- We propose a reinforcement learning technique that exploits partial a priori knowledge

about the system in order to accelerate the rate of learning and improve overall learning performance. Unlike existing reinforcement learning techniques [14], [15], which can only learn about previously visited state-action pairs, the proposed algorithm exploits our partial knowledge about the system's dynamics in order to learn about multiple state-action pairs even before they have been visited.

The remainder of this paper is organized as follows. In Section II, we present the cross-layer problem formulation. In Section III, we describe the two layer multimedia system model. In Section IV, we summarize a well-known reinforcement learning algorithm called Q-learning and we show how it can be used to optimize the system's performance using a centralized optimizer, and how it can be used to optimize the performance of a single layer under the assumption that the other layers deploy static policies. In Section V, we propose two layered (decentralized) Q-learning algorithms for run-time cross-layer optimization. In Section VI, we propose a technique to accelerate the rate of learning by exploiting partial knowledge that we have about the system's structure. In Section VII, we present our experimental results. Finally, we conclude the paper in Section VIII.

II. CROSS-LAYER PROBLEM FORMULATION

In this section, we define the layered structure of the system under study and formulate the cross-layer system optimization problem.

A. *Layered system model*

A typical multimedia system comprises three layers: the *application layer* (APP), the *operating system layer* (OS), and the *hardware layer* (HW). These layers have their own parameters, configurations, and algorithms, which can be adapted at run-time in order to optimize the system's performance (e.g. application quality, power consumption, and delay). We model the layered system as a tuple $\Omega = \langle \mathcal{L}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{S}}, p, R \rangle$, where

- $\mathcal{L} = \{1,...,L\}$ is a set of L autonomous layers, which participate[1] in the cross-layer optimization. Each layer is indexed $l \in \{1,...,L\}$ with layer 1 corresponding to the lowest participating layer (e.g. the HW layer) and layer L corresponding to the highest participating layer (e.g. the APP layer).

[1] We note that for a layer to "participate" in the cross-layer optimization it must be able to adapt one or more of its parameters, configurations, or algorithms (e.g. the APP layer can adapt its source coding parameters); alternatively, if a layer does not "participate", then it is omitted.

- $\boldsymbol{\mathcal{A}}$ is the global action set $\boldsymbol{\mathcal{A}} = \times_{l \in \boldsymbol{\mathcal{L}}} \mathcal{A}_l$ [2], where $\mathcal{A}_l$ is the lth layer's local action set (e.g. the APP layer's available source-coding parameter configurations and the OS layer's available commands for switching the CPU's operating frequency).
- $\boldsymbol{\mathcal{S}}$ is the global state set $\boldsymbol{\mathcal{S}} = \times_{l \in \boldsymbol{\mathcal{L}}} \mathcal{S}_l$, where $\mathcal{S}_l$ is the lth layer's local state set (e.g. the set of buffer states at the APP layer).
- p is the joint transition probability function mapping the global state, global action, and global next-state to a value in $[0,1]$, i.e. $p : \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}} \times \boldsymbol{\mathcal{S}} \mapsto [0,1]$.
- R is the expected reward function, which maps the global state and global action to a real number representing the system's expected reward, i.e. $R : \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}} \mapsto \mathbb{R}_+$.

We assume that the reward function is of the form [9]:

$$R(\boldsymbol{s}, \boldsymbol{a}) = g_L(\boldsymbol{s}, \boldsymbol{a}) - \sum_{l \in \boldsymbol{\mathcal{L}}} \omega_l J_l(s_l, a_l), \tag{1}$$

where J_l is the local cost at layer l, which is independent of the states and actions at the other layers (e.g. the application's rate-distortion cost and hardware's power consumption); the utility gain $g_L(\boldsymbol{s}, \boldsymbol{a})$ measures how well the application at layer L performed given its own state and action, and the states and actions of the other system layers, which support it; and, ω_l weights the relative importance of the cost at layer l with the utility gain. In this paper, we define the utility gain to be closely related to the application's experienced delay (for more details, see Section III).

B. System optimization objective

Unlike existing cross-layer optimization solutions, which focus on the myopic (i.e. immediate) utility, the goal in the proposed cross-layer framework is to find the optimal actions at each stage $n \in \mathbb{N}$ that maximize the *discounted sum of future rewards* [9] [11] [14] [15], i.e.

$$\sum_{n=0}^{\infty} (\gamma)^n R\left(\boldsymbol{s}^n, \boldsymbol{a}^n \mid \boldsymbol{s}^0\right) \tag{2}$$

where the parameter γ $(0 \le \gamma < 1)$ is the "discount factor," which defines the relative importance of present and future rewards, and $\boldsymbol{s}^0$ is the initial state. We refer to the Markov decision policy $\pi^* : \boldsymbol{\mathcal{S}} \mapsto \boldsymbol{\mathcal{A}}$, which maximizes the discounted sum of future rewards from each initial state $\boldsymbol{s}^0 \in \boldsymbol{\mathcal{S}}$, as the optimal *foresighted* policy. We use a *discounted* sum of rewards

[2] $\times_{l \in \mathcal{L}} \mathcal{A}_l = \mathcal{A}_1 \times \cdots \times \mathcal{A}_L$ is the L-ary Cartesian product.

instead of, for example, the *average* reward, because: (i) typical video sources have temporally correlated statistics over short time intervals such that the future environmental dynamics cannot be easily predicted without error [7], and therefore the system may benefit by weighting its immediate reward more heavily than future rewards; and, (ii) The multimedia session's lifetime is not known a priori (i.e. the session may end unexpectedly) and therefore rewards should be maximized sooner rather than later.

Throughout this paper, we will find it useful to work with the optimal *action-value function* $Q^* : \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}} \mapsto \mathbb{R}$ [15]:

$$Q^*(\boldsymbol{s},\boldsymbol{a}) = R(\boldsymbol{s},\boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}' \in \boldsymbol{\mathcal{S}}} p(\boldsymbol{s}' \mid \boldsymbol{s},\boldsymbol{a}) V^*(\boldsymbol{s}'), \quad (3)$$

where $V^*(\boldsymbol{s}) = \max_{\boldsymbol{a}} Q^*(\boldsymbol{s},\boldsymbol{a})$, $\forall \boldsymbol{s} \in \boldsymbol{\mathcal{S}}$, is the optimal *state-value function* [15]. In words, $Q^*(\boldsymbol{s},\boldsymbol{a})$ is the expected sum of discounted rewards achieved by taking action $\boldsymbol{a}$ in state $\boldsymbol{s}$ and then following the optimal policy $\pi^* : \boldsymbol{\mathcal{S}} \mapsto \boldsymbol{\mathcal{A}}$ thereafter, where

$$\pi^*(\boldsymbol{s}) = \arg\max_{\boldsymbol{a}} Q^*(\boldsymbol{s},\boldsymbol{a}),\ \forall \boldsymbol{s} \in \boldsymbol{\mathcal{S}}. \quad (4)$$

III. SYSTEM MODEL

In this section, using the framework introduced above, we model a system in which two layers participate in the cross-layer optimization (i.e. $\mathcal{L} = \{1,2\}$). In particular, the APP layer makes rate-distortion-complexity tradeoffs by adapting its configuration and the OS layer makes energy-delay tradeoffs by adapting the HW layer's operating frequency. Due to the fact that the OS layer controls the HW layer, we combine them into a single decision making layer, which we call the joint OS/HW layer. For clarity, instead of specifying the layers by their indices, in this section we use the subscripts APP to represent quantities related to the APP layer ($l = 2$) and OS to represent quantities related to the joint OS/HW layer's parameters ($l = 1$).

A. *Layer Definitions*

Similar to [21], we model the video source as a sequence of *video data units* (for example, video macroblocks, groups of macroblocks, or pictures), which arrive at a constant rate into the application's pre-encoding buffer. We assume that the system operates synchronously over discrete time slots of variable length, such that one data unit is encoded in each time slot. We interchangeably refer to the time slot during which the nth data unit is encoded as the nth "time

slot" or "stage."

We classify each data unit as being one of Z types in the state set $\mathcal{S}_{\text{APP}}^{(z)} = \{z_i : i = 1,\ldots,Z\}$. The set of states $\mathcal{S}_{\text{APP}}^{(z)}$ depends on the specific video coder being used at the APP layer. In this paper, we assume that $Z = 3$ because video streams are typically compressed into group of pictures structures containing intra-predicted (I), inter-predicted (P), and bi-directionally predicted (B) data units (e.g. MPEG-2, MPEG-4, and H.264/AVC); however, the set of data unit types can be further refined based on, for example, each frame's activity level [7] [20]. It has been shown that transitions among data unit types in an adaptive group of pictures structure can be modeled as a stationary Markov process [20]: i.e.,

$$p_{\text{APP}}^{(z)}(z' \mid z) = \left\{ \Pr(z' \mid z) : z, z' \in \mathcal{S}_{\text{APP}}^{(z)} \right\} \tag{5}$$

where z and z' are the types of the data units that are encoded in time slot n and $n+1$, respectively; and, the probabilities $\Pr(z' \mid z)$ depend on the ratio of I, P, and B data units, the source characteristics, and the condition used to decide when to code an I frame [20].

Each data unit can be coded using any one of H source coding parameter configurations[3] in the APP layer's action set $\mathcal{A}_{\text{APP}} = \{h_i : i = 1,\ldots,H\}$. Given the data unit type $z \in \mathcal{S}_{\text{APP}}^{(z)}$, each configuration $h \in \mathcal{A}_{\text{APP}}$ achieves different operating points in the rate-distortion-complexity space [6]. We let $b(z,h)$, $d(z,h)$, and $c(z,h)$ represent the encoded bit-rate (bits per data unit), incurred distortion (mean square error), and encoding complexity (cycles), respectively. We assume that $b(z,h)$, $d(z,h)$, and $c(z,h)$ are instances of independent and identically distributed random variables.

We penalize the APP layer's actions by employing the Lagrangian cost measure used in the H.264/AVC reference encoder for making rate-distortion optimal mode decisions: i.e. we define the APP layer's cost as

$$J_{\text{APP}}(z,h) = d(z,h) + \lambda_{rd} b(z,h), \tag{6}$$

where $d(z,h)$ is the encoded distortion; $b(z,h)$ is the encoded bit-rate; and $\lambda_{rd} \in [0,\infty)$ is a Lagrangian multiplier, which can be set based on the rate-constraints.

Each data unit is processed at the HW layer's current operating frequency, which is a member of the state set $\mathcal{S}_{\text{OS}} = \{f_i : i = 1,\ldots,F\}$. We let the joint OS/HW layer's cost represent

[3] For example, a video encoder can adapt, at run-time, its choice of quantization parameter, its motion-vector search range, and its choice of motion estimation algorithm.

the power dissipated at operating frequency $f \in \mathcal{S}_{\mathrm{OS}}$, i.e.

$$J_{\mathrm{OS}}(f) = P(f) \text{ (watts)}, \tag{7}$$

where $P(f)$ is the system's power-frequency function. We assume that the OS layer can issue a command in its action set $\mathcal{A}_{\mathrm{OS}} = \{u : u \in \mathcal{S}_{\mathrm{OS}}\}$ to change the HW layer's operating frequency; however, similar to [11], we assume that there is a delay associated with the operating frequency transition. Although this delay can be non-deterministic as described in [11], for simplicity, we assume that there is a deterministic one stage transition delay such that

$$p_{\mathrm{OS}}^{(f)}\left(f' \mid u\right) = \begin{cases} 1, & \text{if } f' = u \\ 0, & \text{otherwise}, \end{cases} \tag{8}$$

where f' is the operating frequency in the next time slot. Note that, regardless of the OS layer's action, the current data unit is processed at the current operating frequency $f \in \mathcal{S}_{\mathrm{OS}}$ at the cost specified in (7).

The APP layer includes a pre-encoding buffer with state set $\mathcal{S}_{\mathrm{APP}}^{(q)} = \{q : i = 0, \ldots, Q\}$, where Q is the maximum number of data units that can be stored in the buffer[4]. The maximum latency for encoding a single data unit without a delay violation is,

$$\text{max delay} = \frac{Q}{\eta} \text{ (seconds)}, \tag{9}$$

where η is the rate (data units per second) at which data units arrive in the pre-encoding buffer. For example, if data units are frames, then η will typically be 15, 24, or 30 frames per second.

The pre-encoding buffer's state at stage $n+1$ can be expressed recursively based on its state at stage n: i.e.,

$$\begin{aligned} q^{n+1} &= \min\left\{\left[q^n + \lfloor t(z^n, h^n, f^n) \cdot \eta \rfloor - 1\right]^+, Q\right\} \\ q^0 &= q_{\mathrm{init}}, \end{aligned} \tag{10}$$

where

$$t(z^n, h^n, f^n) = \frac{c^n(z^n, h^n)}{f^n} \text{ (seconds)} \tag{11}$$

is the nth data unit's processing delay, which depends on its complexity $c^n(z^n, h^n)$ and the processor's operating frequency f^n; $\lfloor x \rfloor$ is the integer part of x; and $[x]^+ = \max\{x, 0\}$. In (10),

[4] In this paper, we assume that Q is not limited by hardware constraints (i.e. available memory); instead, it is determined by the specific application's delay-constraints [21].

the -1 indicates that after the nth data unit is encoded, it departs the pre-encoding buffer. Note that $c^n(z^n,h^n)$ is an instance of the random variable C with distribution $p_C(c \mid z,h)$.

The number of data units that arrive in the pre-encoding buffer during the nth time slot is an instance of the random variable $\tilde{T} = \frac{\eta}{f} \cdot C$ (data units) with distribution

$$\tilde{T} \sim p_{\tilde{T}}(\tilde{t} \mid f,z,h) = \frac{f}{\eta} \cdot p_C\left(\frac{f}{\eta} \cdot c \mid z,h\right) \tag{12}$$

Based on (10) and (12), the pre-encoding buffer's state transition can be modeled as a controllable Markov chain with transition probabilities

$$p_{\text{APP}}^{(q)}(q' \mid q,f,z,h) = \begin{cases} p_{\tilde{T}}\left\{\tilde{t} < 2 \mid f,z,h\right\}, & q = q' = 0 \\ p_{\tilde{T}}\left\{\tilde{t} \geq Q - q + 1 \mid f,z,h\right\}, & q' = Q \\ p_{\tilde{T}}\left\{q' - q + 1 \leq \tilde{t} < q' - q + 2 \mid f,z,h\right\}, & \text{otherwise,} \end{cases} \tag{13}$$

where $p_{\text{APP}}^{(q)}(q' \mid q,f,z,h)$ is the probability that the queue's occupancy transitions from $q \in \mathcal{S}_{\text{APP}}^{(q)}$ (at time slot n) to $q' \in \mathcal{S}_{\text{APP}}^{(q)}$ (at time slot $n+1$) given the operating frequency $f \in \mathcal{S}_{\text{OS}}$, the frame type $z \in \mathcal{S}_{\text{APP}}^{(z)}$, and the APP layer's configuration $h \in \mathcal{A}_{\text{APP}}$.

Given the independent transitions of the data unit's type (see (5)) and the pre-encoding buffer's state (see (13)), the APP layer's transition probability function can be expressed as

$$p_{\text{APP}}\left(\boldsymbol{s}'_{\text{APP}} = (z',q') \mid \boldsymbol{s}_{\text{APP}} = (z,q),f,h\right) = p_{\text{APP}}^{(q)}(q' \mid q,f,z,h) \cdot p_{\text{APP}}^{(z)}(z' \mid z), \tag{14}$$

where $\boldsymbol{s}_{\text{APP}} = (z,q) \in \mathcal{S}_{\text{APP}}$ is the APP layer's state (at time slot n) and $\boldsymbol{s}'_{\text{APP}} = (z',q') \in \mathcal{S}_{\text{APP}}$ is its next state (at time slot $n+1$). Meanwhile, the joint OS/HW layer's transition probability function can be expressed as (see (8))

$$p_{\text{OS}}\left(s'_{\text{OS}} = f' \mid u\right) = p_{\text{OS}}^{(f)}\left(f' \mid u\right). \tag{15}$$

Because the state transitions at the APP layer and joint OS/HW layer are independent, the global transition probability function defined in Section II.A can be factored as

$$p(\boldsymbol{s}' \mid \boldsymbol{s},\boldsymbol{a}) = p_{\text{OS}}\left(s'_{\text{OS}} = f' \mid u\right) p_{\text{APP}}\left(\boldsymbol{s}'_{\text{APP}} = (z',q') \mid \boldsymbol{s}_{\text{APP}} = (z,q),f,h\right). \tag{16}$$

B. System Reward Details

Recall the definition of the reward function in (1). Thus far, we have defined the costs at each layer, but we have not defined the utility gain $g_{\text{APP}}(\boldsymbol{s},\boldsymbol{a})$. As mentioned in Section II.A, the utility gain is closely related to the application's experienced queuing delay in the pre-encoding buffer. Conventionally, multimedia buffers are used to enforce a maximum tolerable processing

[22] or transmission delay [21] (e.g. our buffer imposes a maximum processing delay of $\frac{Q}{\eta}$ seconds, as defined in (9)) and as long as the buffer does not overflow (i.e. the overflow constraint is not violated), there is no penalty. In our setting, however, the multimedia system's dynamics are unknown and non-stationary; therefore, we cannot guarantee that the buffer overflow constraints will be satisfied. In light of this, we design the utility gain to non-linearly reward the system for maintaining queuing delays less than the maximum tolerable delay, thereby protecting against overflows that may result from a sudden increase in encoding delay. Formally, we define the utility gain as

$$g_{\text{APP}}(\boldsymbol{s}, \boldsymbol{a}) = 1 - \left(\frac{q + \lfloor \tilde{t} \rfloor - 1}{Q} \right)^2, \tag{17}$$

which is near its maximum when the buffer is empty (corresponding to zero queuing delay) and is minimized when the buffer is full (corresponding to the maximum tolerable queuing delay $\frac{Q}{\eta}$). In Appendix A, we experimentally show that (17) is a good definition for the utility gain.

Given the utility gain defined in (17), the APP layer's cost defined in (6), and the joint OS/HW layer's cost defined in (7), the stage reward defined in (1) can be rewritten as

$$R(\boldsymbol{s}, \boldsymbol{a}) = 1 - \left(\frac{q + \lfloor \tilde{t} \rfloor - 1}{Q} \right)^2 - \omega_{\text{OS}} P(f) - \omega_{\text{APP}} \left[d(z, h) + \lambda_{rd} b(z, h) \right] \tag{18}$$

IV. LEARNING THE OPTIMAL DECISION POLICY

A. *Why Model-Free Learning?*

As we mentioned before, the multimedia system's dynamics (i.e. $R(\boldsymbol{s}, \boldsymbol{a})$ and $p(\boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a})$) are *unknown* and therefore the optimal action-value function Q^* and the optimal policy π^* must be learned online, based on experience. However, it remains to explain how we can learn the optimal policy online despite these unknown quantities. Let us consider the following two options:

- *Direct estimation and policy computation:* One option is to directly estimate the reward and transition probability function using, for example, maximum likelihood estimation, and then to perform value iteration [15] to determine the optimal policy; however, this

solution far too complex.

- *Offline policy computation and online policy interpolation:* To avoid using the complex value iteration algorithm online, in [12] the authors propose a two step procedure involving an offline stage and an online stage. Offline, they quantize every unknown probability into NS samples (more samples yield more accurate results). For each quantized probability bin, they compute a corresponding policy. Then, online, they directly estimate the transition probability function and, based on this, interpolate the offline computed policies to determine the current policy. For M parameters, this learning algorithm requires $\prod_{m=1}^{M} NS_m$ policy look-up tables of size $|\boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}}|$ as defined in [11]. However, if there are many unknown parameters, this requires a huge amount of memory.

Because neither of the above solutions are practical in our resource-constrained cross-layer setting with many states and actions (and many unknown parameters), we adopt a *model-free* reinforcement learning solution, which can be used to learn the optimal action-value function Q^* and the corresponding optimal policy π^* online without directly estimating the reward and transition probability functions. Specifically, we adopt a low complexity algorithm called Q-learning, which is also light on memory. In our cross-layer setting, Q-learning can be implemented by a central optimizer -- located at either the APP layer, the joint OS/HW layer, or a separate middleware layer -- to learn the globally optimal policy online. Alternatively, a single layer can use Q-learning to learn its *local best-response policy* online, under the constraint that the other system layers deploy a stationary (static) policy. We discuss the former case in Subsection IV.B and the latter case in Subsection IV.C.

B. Globally Optimal Centralized Q-learning

Central to the globally optimal centralized Q-learning algorithm is a simple update step performed at the end of each time slot based on the *experience tuple* (ET) $\left(\boldsymbol{s}^n, \boldsymbol{a}^n, r^n, \boldsymbol{s}^{n+1}\right)$:

$$\delta^n = \left[r^n + \gamma \max_{\boldsymbol{a}' \in \boldsymbol{\mathcal{A}}} Q^n(\boldsymbol{s}^{n+1}, \boldsymbol{a}')\right] - Q^n(\boldsymbol{s}^n, \boldsymbol{a}^n), \tag{19}$$

$$Q^{n+1}(\boldsymbol{s}^n, \boldsymbol{a}^n) \leftarrow Q^n(\boldsymbol{s}^n, \boldsymbol{a}^n) + \alpha^n \delta^n, \tag{20}$$

where $\boldsymbol{s}^n$, $\boldsymbol{a}^n$, and $r^n = g_L^n - \sum_{l \in \boldsymbol{\mathcal{L}}} \omega_l J_l^n$ are the state, performed action, and corresponding reward in time slot n, respectively; $\boldsymbol{s}^{n+1}$ is the resulting state in time slot $n+1$; $\boldsymbol{a}'$ is the *greedy*

action[5] in state $\boldsymbol{s}^{n+1}$; δ^n is the so-called *temporal-difference* (TD) error [14]; and, $\alpha^n \in [0,1]$ is a time-varying learning rate parameter. We note that the action-value function is conventionally initialized arbitrarily at time $n = 0$.

It is well known that if (i) the rewards and transition probability functions are stationary, (ii) all of the state-action pairs are visited infinitely often, and (iii) α^n satisfies the *stochastic approximation conditions*[6] $\sum_{n=0}^{\infty}(\alpha^n) = \infty$ and $\sum_{n=0}^{\infty}(\alpha^n)^2 < \infty$, then Q converges with probability 1 to Q^* [16]. Subsequently, the optimal policy can be found using (4). In the considered multimedia system, however, the dynamics are unknown and non-stationary (e.g. due to changes in the source characteristics). Therefore, convergence cannot be guaranteed. Under non-stationary conditions, it is prudent to set α^n to a small constant in order to track the experienced dynamics.

During the learning process, it is not entirely obvious what the best action is to take in each state. On the one hand, the optimal action-value function can be learned by randomly *exploring* the available actions in each state. Unfortunately, unguided randomized exploration cannot guarantee acceptable performance during the learning process. On the other hand, taking greedy actions, which *exploit* the available information in the action-value function $Q(\boldsymbol{s},\boldsymbol{a})$, can guarantee a certain level of performance. Unfortunately, exploiting what is already known about the system prevents the discovery of new, better, actions. To judiciously trade off exploration and exploitation, we use the so-called ε*-greedy* action selection method [14]:

ε***-greedy* action selection:** With probability $1-\varepsilon$, take the greedy action that maximizes the action-value function, i.e. $\boldsymbol{a}^* = \arg\max_{\boldsymbol{a}}\{Q(\boldsymbol{s},\boldsymbol{a})\}$; and, with probability ε, take an action randomly and uniformly over the action set.

We write $\boldsymbol{a} = \Phi_\varepsilon(Q,\boldsymbol{s})$ to show that $\boldsymbol{a}$ is an ε*-greedy* joint-action and $a_l = [\Phi_\varepsilon(Q,\boldsymbol{s})]_l$ to represent the component of the ε*-greedy* action at layer l, where $\boldsymbol{a} = (a_1,\ldots,a_L)$. If the dynamics are stationary, then ε must satisfy the same stochastic approximation conditions as the learning rate α to ensure that the learned decision policy converges to the optimal policy.

[5] A greedy action is one that maximizes the current estimate of the action-value function, i.e. $\boldsymbol{a}^* = \arg\max_{\boldsymbol{a}}\{Q(\boldsymbol{s},\boldsymbol{a})\}$.

[6] For example, $\alpha^n = 1/(n+1)$ satisfies the stochastic approximation conditions.

Fig. 1 illustrates the information exchanges required to deploy the centralized Q-learning algorithm in a two layer multimedia system during one time slot. In Fig. 1, the top and bottom blocks represent the APP layer and joint OS/HW layer, respectively. The center block represents the centralized optimizer, which selects both layers' actions and updates the system's global action-value function Q. As we mentioned before, the centralized optimizer may be located at either the APP layer, the joint OS/HW layer, or a separate middleware layer. To best highlight the information exchanges, we illustrate the centralized optimizer as a separate middleware layer. Although we do not explicitly indicate this in Fig. 1, the ET $(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')$ is used by the block labeled "Update $Q(\boldsymbol{s}, \boldsymbol{a})$," which performs the update step defined in (20).

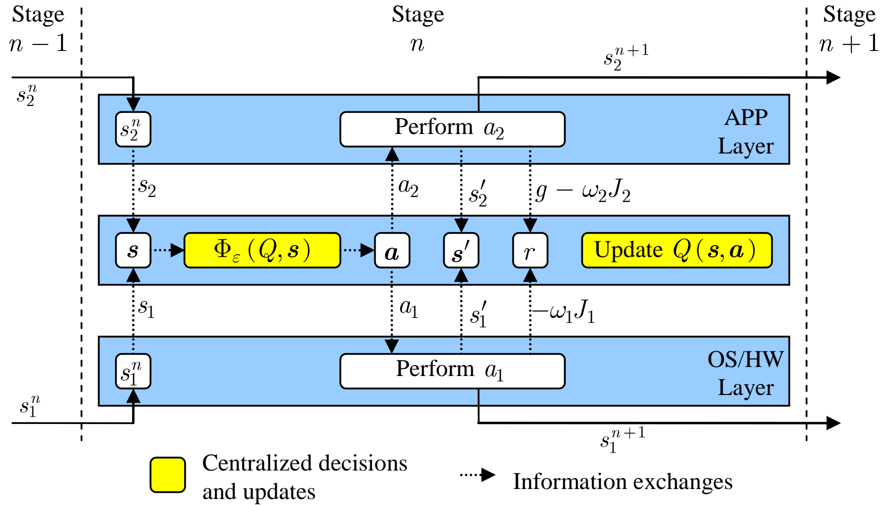

Fig. 1. Information exchanges required to deploy the centralized Q-learning update step in one time slot.

C. *Proposed Local Best-Response Q-learning*

As we mentioned earlier, a single layer can use Q-learning to learn its local best-response policy online, under the constraint that the other system layers deploy a fixed policy.

Let $\pi = (\pi_1, \ldots, \pi_L)$, where π_l is a mapping from the global state to the lth layer's local action, i.e. $\pi_l : \boldsymbol{\mathcal{S}} \mapsto \mathcal{A}_l$. Let $\boldsymbol{\mathcal{L}}_{-l} = (1, \ldots, l-1, l+1, \ldots, L)$ be the set of all layers excluding layer l and let $\pi_{-l} = (\pi_1, \ldots, \pi_{l-1}, \pi_{l+1}, \ldots, \pi_L)$ be the joint decision policy of all the layers in $\boldsymbol{\mathcal{L}}_{-l}$. If π_{-l} is fixed, then the stage reward $R(\boldsymbol{s}, \boldsymbol{a})$ defined in (18) can be rewritten as

$$R(\boldsymbol{s}, a_l \mid \pi_{-l}) = R(\boldsymbol{s}, (a_l, \pi_{-l}(\boldsymbol{s}))), \tag{21}$$

and the transition probability function defined in (16) can be rewritten as

$$p(\boldsymbol{s}' \mid \boldsymbol{s}, a_l, \pi_{-l}) = p(\boldsymbol{s}' \mid \boldsymbol{s}, (a_l, \pi_{-l}(\boldsymbol{s}))), \tag{22}$$

such that the only variable in both of these quantities is the action taken by layer l, i.e. a_l. Note

that, although the local costs at layers $l' \in \mathcal{L}_{-l}$ (i.e. $J_{l'}(s_{l'}, \pi_{l'}(s_{l'}))$) are independent of the lth layer's action a_l, layer l still needs to know them. This is because each layer needs to learn how its actions impact the global system reward and not just its local costs [17]. For instance, if the joint OS/HW layer is unaware of its impact on the application's delay and quality it will always selfishly minimize its own costs by operating at its lowest frequency and power. Based on this observation, we define the best-response action-value function at layer l as

$$Q_l^*(\boldsymbol{s}, a_l \mid \pi_{-l}) = R(\boldsymbol{s}, a_l \mid \pi_{-l}) + \gamma \sum_{\boldsymbol{s}' \in \boldsymbol{\mathcal{S}}} p(\boldsymbol{s}' \mid \boldsymbol{s}, a_l, \pi_{-l}) \max_{a_l \in \mathcal{A}_l} Q_l^*(\boldsymbol{s}', a_l \mid \pi_{-l}), \tag{23}$$

Then, the local best-response policy at layer l, which we denote by $\pi_l^*(\boldsymbol{s} \mid \pi_{-l})$, can be computed as

$$\pi_l^*(\boldsymbol{s} \mid \pi_{-l}) = \arg\max_{a_l \in \mathcal{A}_l} Q_l^*(\boldsymbol{s}, a_l \mid \pi_{-l}). \tag{24}$$

The lth layer's locally optimal action-value function Q_l^* and the corresponding local best-response policy $\pi_l^*(\boldsymbol{s} \mid \pi_{-l})$ can be learned online using a straightforward adaptation of the centralized Q-learning update step described in the previous subsection: i.e.,

$$\delta_l^n = \left[r^n + \gamma \max_{a_l' \in \mathcal{A}_l} Q_l^n(\boldsymbol{s}^{n+1}, a_l' \mid \pi_{-l})\right] - Q_l^n(\boldsymbol{s}^n, a_l^n \mid \pi_{-l}), \tag{25}$$

$$Q_l^{n+1}(\boldsymbol{s}^n, a_l^n \mid \pi_{-l}) \leftarrow Q_l^n(\boldsymbol{s}^n, a_l^n \mid \pi_{-l}) + \alpha_l^n \delta_l^n, \tag{26}$$

where $r^n = g_L^n - \sum_{l=1}^{L} \omega_l J_l^n$ is a random sample of the reward with expected value $R(\boldsymbol{s}^n, a_l^n \mid \pi_{-l})$. Unlike the global Q-learning update defined in (20), which uses the experience tuple $(\boldsymbol{s}^n, \boldsymbol{a}^n, r^n, \boldsymbol{s}^{n+1})$, the best response Q-learning update at layer l requires the experience tuple $(\boldsymbol{s}^n, a_l^n, r^n, \boldsymbol{s}^{n+1})$, which only includes its local action a_l^n instead of the global action $\boldsymbol{a}^n$. Note that, given the global state $\boldsymbol{s}^n$, layer l trades off exploitation and exploration by selecting the ε-greedy action $a_l^n = \Phi_\varepsilon(Q_l^n, \boldsymbol{s}^n)$.

Fig. 2 illustrates the information exchanges required to deploy the local best-response Q-learning algorithm in a two layer multimedia system during one time slot. Fig. 2(a) shows the algorithm at the APP layer and Fig. 2(b) shows it at the joint OS/HW layer. Although we do not explicitly indicate this in Fig. 2(a) or Fig. 2(b), the ETs $(\boldsymbol{s}, a_l, r, \boldsymbol{s}')$, $l \in \{1,2\}$, are used by the blocks labeled "Update $Q_l(\boldsymbol{s}, a_l \mid \pi_{-l})$," which perform the update step defined in (26). Note that the update step is not performed at layer $-l$, because it deploys the static policy π_{-l}.

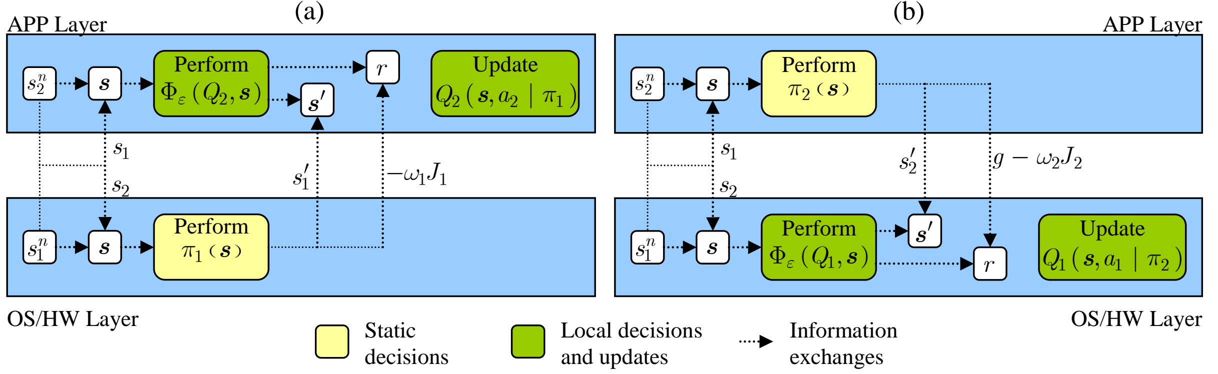

Fig. 2. Information exchanges required to deploy the local best-response Q-learning update step in one time slot. (a) APP layer deploys local best-response Q-learning while the joint OS/HW layer deploys a static policy. (b) Joint OS/HW layer deploys local best-response Q-learning while the APP layer deploys a static policy.

V. AUTONOMOUS LAYERED LEARNING

Thus far, we have discussed how Q-learning can be implemented by a centralized optimizer or by a single layer in order to learn the optimal global policy or the local best-response policy online. Unfortunately, the former Q-learning algorithm violates the layered system architecture and the latter does not solve the cross-layer optimization (i.e. it is only a single-layer optimization). In this section, we propose two autonomous layered learning algorithms that are closely related to the two aforementioned learning solutions. First, in subsection V.A, we discuss how the APP layer and the joint OS/HW layers can coordinate their local best-response learning algorithms to improve the system's performance over time. Second, in subsection V.B, we describe how the global action-value function can be decomposed based on the structure of the transition probability and reward functions, and how this decomposition leads to a layered Q-learning algorithm. In subsection V.C, we compare the computation, communication, and memory overheads associated with the centralized, coordinated best-response, and layered Q-learning algorithms.

A. Proposed Coordinated Best-Response Q-Learning

Recall from subsection IV.C that $\pi_l^*(s \mid \pi_{-l})$ is the local best-response policy at layer l given the static policy π_{-l} at layer $-l$. In this subsection, we propose a layered learning technique that attempts to determine the globally optimal action-value function Q^* and policy π^* by alternating between local best-response learning at the APP layer, while the joint OS/HW layer maintains a static policy, and vice versa. In other words, we alternate between the local

best-response learning algorithms illustrated in Fig. 2(a) and Fig. 2(b).

To coordinate the local best-response learning processes at the two layers, we define the *coordination function* $\chi : \{0,1,2,\ldots\} \to \mathcal{L}$: if $\chi(n) = l$, then layer l can learn in time slot n and layer $-l$ must maintain a static policy. For illustration, in our experiments we restrict ourselves to a class of policies that can be characterized by the pair (N_1, N_2), and are defined as follows:

$$\chi(n) = \begin{cases} 1, & \text{if } [n]_{N_1+N_2} < N_1 \\ 2, & \text{otherwise,} \end{cases} \tag{27}$$

where $[X]_N$ denotes X modulo N. In other words, the class of (N_1, N_2)*-coordination policies* allows layer $l = 1$ to learn for N_1 time slots, and then for layer $l = 2$ to learn for N_2 time slots, and then for layer $l = 1$ to learn for N_1 time slots again, etc. This class of coordination policies is desirable due to its simplicity: prior to execution, each layer can initialize N_1 and N_2, then, at run-time, they can autonomously coordinate themselves using (27) without any additional synchronization overheads. In our experiments, we investigate the impact of different (N_1, N_2)-coordination policies on the system's learning performance.

An obvious drawback of this algorithm is that only one layer can learn in each time slot. Consequently, the learning algorithm adapts slowly to the environment and it is not guaranteed to converge to the optimal global policy (i.e. achieving the optimal policy may require both layers to simultaneously adapt their actions). In the next subsection, we propose a layered Q-learning algorithm that allows both layers to learn simultaneously, thereby improving the system's learning performance.

B. Proposed Layered Q-learning

In this subsection, we first show how the action-value function can be decomposed given the structure of the cross-layer optimization. Subsequently, we show how this decomposition leads to a layered learning algorithm, which solves the cross-layer optimization *online*, in a decentralized manner, when the transition probability and reward functions are *unknown* a priori.

1) Action-Value Function Decomposition

Given the additive reward function defined in (18) and the factored transition probability function defined in (16), we can rewrite the optimal action-value function defined in (3) as follows:

$$Q^*(\boldsymbol{s},\boldsymbol{a}) = g_2(\boldsymbol{s},a_2) - \omega_1 J_1(s_1,a_1) - \omega_2 J_2(s_2,a_2) + \gamma \sum_{s_1' \in \mathcal{S}_1, s_2' \in \mathcal{S}_2} p_1(s_1' \mid s_1,a_1) p_2(s_2' \mid \boldsymbol{s},a_2) V^*(\boldsymbol{s}'), \tag{28}$$

where, $V^*(\boldsymbol{s}') = \max_{\boldsymbol{a}' \in \boldsymbol{\mathcal{A}}} Q^*(\boldsymbol{s}',\boldsymbol{a}')$ is the optimal state-value for state $\boldsymbol{s}'$. In (28), the subscripts 1 and 2 represent the joint OS/HW layer and APP layer, respectively; $\boldsymbol{s} = (s_1, s_2)$ is the global state comprising the local states $s_1 = f$ (operating frequency) and $s_2 = (z, q)$ (data unit type and buffer state, respectively); $\boldsymbol{a} = (a_1, a_2)$ is the global action comprising the local actions $a_1 = u$ (frequency request) and $a_2 = h$ (application configuration); $p_1(s_1' \mid s_1, a_1)$ and $\omega_1 J_1(s_1, a_1)$ are the joint OS/HW layer's local transition probability function and local weighted reward function, respectively; $p_2(s_2' \mid \boldsymbol{s}, a_2)$ and $\omega_2 J_2(s_2, a_2)$ are the APP layer's local transition probability function and local weighted reward function, respectively; and, $g_2(\boldsymbol{s}, a_2)$ is the utility gain function.

Observing that $g_2(\boldsymbol{s},a_2) - \omega_2 J_2(s_2,a_2)$ is independent of s_1' and that $\sum_{s_1' \in \mathcal{S}_1} p_1(s_1' \mid s_1, a_1) = 1$, we may rewrite (28) as follows:

$$Q^*(\boldsymbol{s},a_1,a_2) = -\omega_1 J_1(s_1,a_1) + \sum_{s_1' \in \mathcal{S}_1} \left[p_1(s_1' \mid s_1,a_1) \begin{pmatrix} g_2(\boldsymbol{s},a_2) - \omega_2 J_2(s_2,a_2) + \\ \gamma \sum_{s_2' \in \mathcal{S}_2} p_2(s_2' \mid \boldsymbol{s},a_2) V^*(s_1', s_2') \end{pmatrix} \right]. \tag{29}$$

Given the global state $\boldsymbol{s} = (s_1, s_2)$ and the optimal state-value function V^*, the APP layer can perform the inner computation:

$$Q_1^*(\boldsymbol{s},a_2,s_1') = \begin{pmatrix} g_2(\boldsymbol{s},a_2) - \omega_2 J_2(s_2,a_2) + \\ \gamma \sum_{s_2' \in \mathcal{S}_2} p_2(s_2' \mid \boldsymbol{s},a_2) V^*(s_1', s_2') \end{pmatrix}, \ \forall a_2 \in \mathcal{A}_2 \text{ and } \forall s_1' \in \mathcal{S}_1 \tag{30}$$

which is independent of the immediate action at the joint OS/HW layer (i.e. a_1), but depends on the joint OS/HW layer's potential next-state s_1'[7]. $Q_1^*(\boldsymbol{s},a_2,s_1')$ can be interpreted as the APP layer's estimate of the expected discounted future rewards at the joint OS/HW layer. Hence, for each global state $\boldsymbol{s} = (s_1, s_2)$ the APP layer must compute the set $\left\{ Q_1^*(\boldsymbol{s},a_2,s_1') : a_2 \in \mathcal{A}_2, s_1' \in \mathcal{S}_1 \right\}$ using (30). Then, given this set from the APP layer, the joint

OS/HW layer can perform the outer computation in (29): i.e.,

$$Q^*(\boldsymbol{s}, a_1, a_2) = \left(\begin{array}{l} -\omega_1 J_1(s_1, a_1) + \\ \sum_{s_1' \in \mathcal{S}_1} p_1(s_1' \mid s_1, a_1) Q_1^*(\boldsymbol{s}, a_2, s_1') \end{array} \right), \ \forall a_1 \in \mathcal{A}_1 \text{ and } \forall a_2 \in \mathcal{A}_2 \tag{31}$$

Q^* can be computed by repeating this procedure for all $\boldsymbol{s} \in \boldsymbol{\mathcal{S}}$.

For more information about this decomposition, we refer the interested reader to our prior paper [9], in which we use a similar decomposition to solve the cross-layer optimization *offline*, in a decentralized manner, under the assumption that the transition probability and reward functions are *known* a priori.

2) Layered Q-learning

Recall that the centralized Q-learning algorithm described in Section IV.B violates the layered system architecture because it requires a centralized manager to select actions for both of the layers. In contrast, the layered Q-learning algorithm that we propose in this subsection – made possible by the action-value function decomposition described above – adheres to the layered architecture by enabling each layer to autonomously select its own actions and to update its own action-value function. In the following, we discuss the local information requirements for each layer, how each layer selects its local actions, and how each layer updates its local action-value function.

Local information: The proposed layered Q-learning algorithm requires that the APP layer maintains an estimate of the action-value function on the left-hand side of (30): i.e.,

$$\left\{ Q_1(\boldsymbol{s}, a_2, s_1') : \boldsymbol{s} \in \boldsymbol{\mathcal{S}},\ a_2 \in \mathcal{A}_2,\ s_1' \in \mathcal{S}_1 \right\},$$

and that the joint OS/HW layer maintains an estimate of the action-value function on the left-hand side of (31): i.e.,

$$\left\{ Q(\boldsymbol{s}, a_1, a_2) : \boldsymbol{s} \in \boldsymbol{\mathcal{S}}, (a_1, a_2) \in \boldsymbol{\mathcal{A}} \right\}.$$

Local action selection: At run-time, given the current global state $\boldsymbol{s} = (s_1, s_2)$, the APP layer selects an ε-greedy action $a_2 \in \mathcal{A}_2$ as described in Section IV.B, but with the greedy action selected as follows:

$$a_2^* = \arg\max_{a_2 \in \mathcal{A}_2} \left\{ \max_{s_1' \in \mathcal{S}_1} Q_1(\boldsymbol{s}, a_2, s_1') \right\}. \tag{32}$$

---

[7] Although the model in (8) assumes a deterministic relationship between the joint OS/HW layer's action a_1 and its next-state s_1', this is not required for the decomposition to work.

Note that action a_2^* is selected under the assumption that the joint OS/HW layer will select its action to transition to the best next-state. Then, given action a_2^* from the APP layer, the joint OS/HW layer selects ε-greedy action $a_1 \in \mathcal{A}_1$, but with the greedy action selected as follows:

$$a_1^* = \arg\max_{a_1 \in \mathcal{A}_1} \{ Q_1(\boldsymbol{s}, a_1, a_2^*) \}. \quad (33)$$

Local learning updates: After executing the ε-greedy action $\boldsymbol{a}^n = (a_1^n, a_2^n)$ in state $\boldsymbol{s}^n = (s_1^n, s_2^n)$, the system obtains the reward $r^n = g_2^n - \omega_1 J_1^n - \omega_2 J_2^n$ and transitions to state $\boldsymbol{s}^{n+1} = (s_1^{n+1}, s_2^{n+1})$. Based on the experience tuple $(\boldsymbol{s}^n, \boldsymbol{a}^n, r^n, \boldsymbol{s}^{n+1})$, each layer updates its action-value function as follows. First, the joint OS/HW layer must forward the scalar $V^n(s_1^{n+1}, s_2^{n+1}) = \max_{a_1' \in \mathcal{A}_1, a_2' \in \mathcal{A}_2} Q^n(s_1^{n+1}, s_2^{n+1}, a_1', a_2')$ to the APP layer. Using this forwarded information, the APP layer can perform its action-value function update based on the form of (30): i.e.,

$$\delta_2^n = \left[g_2^n - \omega_2 J_2^n + \gamma V^n(s_1^{n+1}, s_2^{n+1}) \right] - Q_1^n(\boldsymbol{s}^n, a_2^n, s_1^{n+1}), \quad (34)$$

$$Q_1^{n+1}(\boldsymbol{s}^n, a_2^n, s_1^{n+1}) \leftarrow Q_1^n(\boldsymbol{s}^n, a_2^n, s_1^{n+1}) + \alpha_2^n \delta_2^n. \quad (35)$$

Then, given the scalar $Q_1^{n+1}(\boldsymbol{s}^n, a_2^n, s_1^{n+1})$ from the APP layer and the APP layer's selected action a_2^n, the joint OS/HW layer can perform its action-value function update based on the form of (31) as follows:

$$\delta_1^n = \left[-\omega_1 J_1^n + Q_1^{n+1}(\boldsymbol{s}^n, a_2^n, s_1^{n+1}) \right] - Q^n(\boldsymbol{s}^n, a_1^n, a_2^n), \quad (36)$$

$$Q^{n+1}(\boldsymbol{s}^n, a_1^n, a_2^n) \leftarrow Q^n(\boldsymbol{s}^n, a_1^n, a_2^n) + \alpha_1^n \delta_1^n. \quad (37)$$

An added benefit of the decentralized Q-learning algorithm over the coordinated best-response algorithm is that layers do not need to directly share their local rewards (i.e. the local learning updates only require the APP layer to know its local reward $g_2^n - \omega_2 J_2^n$ and for the joint OS/HW layer to know its local cost $\omega_1 J_1^n$). This is important if the company designing a layer wants this information to remain private in order to protect the underlying intellectual property.

Fig. 3 illustrates the information exchanges required to deploy the layered Q-learning algorithm in a two layer multimedia system during one time slot. Although we do not explicitly indicate this in Fig. 3, the ETs $(\boldsymbol{s}, a_2, g_2 - \omega_2 J_2, s_1')$ and $(\boldsymbol{s}, a_1, -\omega_1 J_1, s_1')$ are used by the blocks labeled "Update $Q_1(\boldsymbol{s}, a_2, s_1')$" at the APP layer and "Update $Q(\boldsymbol{s}, a_1, a_2)$" at the joint OS/HW layer, respectively.

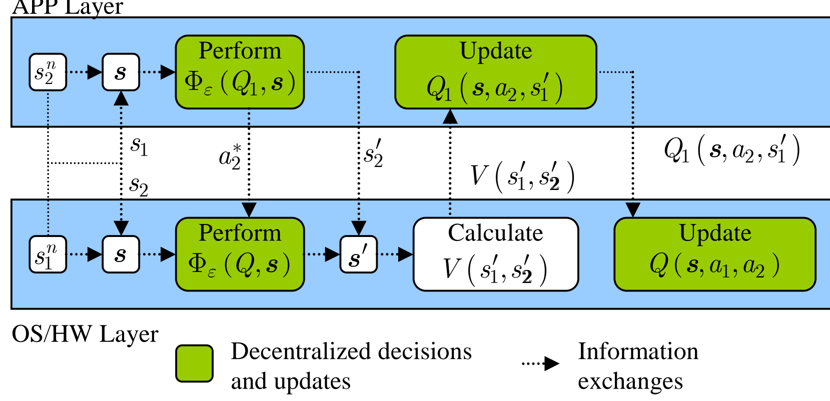

Fig. 3. Information exchanges required to deploy the layered Q-learning update step in one time slot.

C. Computation, communication, and memory overheads

In this subsection, we compare the computation, communication, and memory overheads associated with the centralized, coordinated best-response, and layered Q-learning algorithms. Table I summarizes the greedy action selection procedure and update steps associated with each algorithm; and, Table II lists the computation, communication, and memory overheads associated with each algorithm.

From Table II, we observe that the coordinated best-response algorithm has the least computation, communication, and memory overheads. This is because the layers do not learn at the same time so they can ignore the other layer's actions. Meanwhile, the layered Q-learning algorithm is the most complex and requires the most memory. Note that, in our setting, $\overline{\overline{\mathcal{S}_1}} = \overline{\overline{\mathcal{A}_1}}$; hence, the decentralized Q-learning algorithm incurs over twice the computational overheads and exactly twice the memory overheads as the centralized algorithm. The increased overheads can be interpreted as the cost of optimal decentralized learning (optimal in the sense that it performs as well as the centralized learning algorithm). We also observe that the communication overheads in all cases are $O(1)$ because they are independent of the size of the state sets and action sets at each layer; however, the precise number of messages exchanged between the two layers depends on the deployed learning algorithm as illustrated in Fig. 1, Fig. 2, and Fig. 3, and noted in Table II.

Table I. Summary of greedy action selection and update steps for various Q-learning algorithms.

	Greedy Action	Update Step
Centralized Q-learning	Middleware Layer: $\boldsymbol{a}^* = \arg\max_{\boldsymbol{a} \in \boldsymbol{\mathcal{A}}} \{Q(\boldsymbol{s}, \boldsymbol{a})\}$	Middleware Layer: $\delta^n = \left[r^n + \gamma \max_{\boldsymbol{a}' \in \boldsymbol{\mathcal{A}}} Q(\boldsymbol{s}^{n+1}, \boldsymbol{a}')\right] - Q(\boldsymbol{s}^n, \boldsymbol{a}^n)$ $Q(\boldsymbol{s}^n, \boldsymbol{a}^n) \leftarrow Q(\boldsymbol{s}^n, \boldsymbol{a}^n) + \alpha^n \delta^n$

Coordinated best-response Q-learning	Layers $l \in \{1,2\}$: $a_l^* = \arg\max_{a_l \in \mathcal{A}_l} Q_l^*(\boldsymbol{s}, a_l \mid \pi_{-l})$	Layers $l \in \{1,2\}$: $\delta_l^n = \left\{ \begin{matrix} \left[r^n + \gamma \max_{a_l' \in \mathcal{A}_l} Q_l(\boldsymbol{s}^{n+1}, a_l' \mid \pi_{-l}) \right] \\ -Q_l(\boldsymbol{s}^n, a_l^n \mid \pi_{-l}) \end{matrix} \right\}$ $Q_l(\boldsymbol{s}^n, a_l^n \mid \pi_{-l}) \leftarrow Q_l(\boldsymbol{s}^n, a_l^n \mid \pi_{-l}) + \alpha_l^n \delta_l^n$
Layered Q-learning	Layer $l = 1$: $a_1^* = \arg\max_{a_1 \in \mathcal{A}_1} \{ Q_1(\boldsymbol{s}, a_1, a_2^*) \}$ Layer $l = 2$: $a_2^* = \arg\max_{a_2 \in \mathcal{A}_2} \left\{ \max_{s_1' \in \mathcal{S}_1} Q_1(\boldsymbol{s}, a_2, s_1') \right\}$	Layer $l = 1$: $\delta_1^n = \left\{ \begin{matrix} \left[-\omega_1 J_1^n + Q_1^{n+1}(\boldsymbol{s}^n, a_2^n, s_1^{n+1}) \right] \\ -Q^n(\boldsymbol{s}^n, a_1^n, a_2^n) \end{matrix} \right\}$ $Q^{n+1}(\boldsymbol{s}^n, a_1^n, a_2^n) \leftarrow Q^n(\boldsymbol{s}^n, a_1^n, a_2^n) + \alpha_1^n \delta_1^n$ Layer $l = 2$: $\delta_2^n = \left\{ \begin{matrix} \left[g_2^n - \omega_2 J_2^n + \gamma V^n(s_1^{n+1}, s_2^{n+1}) \right] \\ -Q_1^n(\boldsymbol{s}^n, a_2^n, s_1^{n+1}) \end{matrix} \right\}$ $Q_1^{n+1}(\boldsymbol{s}^n, a_2^n, s_1^{n+1}) \leftarrow Q_1^n(\boldsymbol{s}^n, a_2^n, s_1^{n+1}) + \alpha_2^n \delta_2^n$

Table II. Comparison of computation, communication, and memory overheads.

	Centralized Q-learning	Coordinated best-response Q-learning	Layered Q-learning
Computation overheads	Action Selection: $O\left(\overline{\overline{\boldsymbol{\mathcal{A}}}}\right)$ Update: $O\left(\overline{\overline{\boldsymbol{\mathcal{A}}}}\right)$	Action Selection: $O\left(\overline{\overline{\mathcal{A}_1}} + \overline{\overline{\mathcal{A}_2}}\right)$ Update (layer l): $O\left(\overline{\overline{\mathcal{A}_l}}\right)$	Action Selection: $O\left(\overline{\overline{\mathcal{A}_1}} + \overline{\overline{\mathcal{S}_1 \times \mathcal{A}_2}}\right)$ Update: $O\left(\overline{\overline{\boldsymbol{\mathcal{A}}}}\right)$
Communication overheads	$O(1)$ (8 messages)	$O(1)$ (4 messages)	$O(1)$ (6 messages)
Memory overheads	$O\left(\overline{\overline{\boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}}}}\right)$	$O\left(\overline{\overline{\boldsymbol{\mathcal{S}} \times \mathcal{A}_1}}\right) + O\left(\overline{\overline{\boldsymbol{\mathcal{S}} \times \mathcal{A}_2}}\right)$	$O\left(\overline{\overline{\boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}}}} + \overline{\overline{\boldsymbol{\mathcal{S}} \times \mathcal{A}_2 \times \mathcal{S}_1}}\right)$

VI. ACCELERATED LEARNING USING PARTIAL KNOWLEDGE

In conventional reinforcement learning [14] [15], the actors (i.e. the system layers in our setting) are assumed to have no a priori information about the form of the transition probability and reward functions beyond possible high-level structural knowledge about the factored transition and reward dynamics [17] [23]. In other words, in a conventional reinforcement learning framework, the additive decomposition structure of the reward function defined in (1) and the factored transition probability structure defined in (14) may be known a priori, but the actual form of the utility gain defined in (17) and the actual form of the transition probability function defined in (13) and (8) cannot be known a priori. As system designers, however, we have knowledge about the form of these functions, which we can exploit when designing a learning algorithm.

There are two points in the Q-learning algorithm where we can exploit our partial knowledge in order to dramatically improve the speed of learning. The first point is through the update step: in subsection VI.A, we discuss how we can use the experience encapsulated in one ET in order to update multiple *statistically equivalent* state-action pairs. The second point is during initialization: in subsection VI.B, we show how we can initialize the action-value function in time slot $n = 0$.

A. Accelerated Learning Using Virtual Experience Tuples

The Q-learning update step defined in (20) updates the action-value function for only one state-action pair in each time slot. Consequently, in stationary environments, Q-learning takes a large number of stages before Q converges to Q^*; and, in non-stationary environments, Q-learning does not adapt quickly to changes. Several existing variants of Q-learning adapt the action-value function for multiple state-action pairs in each time slot. These include temporal-difference-λ updates, Dyna, and prioritized sweeping [14] [15]; however, these solutions are not system specific and assume no a priori knowledge of the problem's structure. Consequently, these existing solutions can only update previously visited state-action pairs.

In this subsection, we propose a new Q-learning variant, which is complementary to the abovementioned variants. The proposed algorithm exploits the *form* of the transition probability and reward functions (defined in Section III) by updating the action-value function for multiple *statistically equivalent* state-action pairs in each time slot, including those that have never been visited. In stationary (non-stationary) environments, the proposed algorithm improves convergence time (adaptation speed) at the expense of increased computational complexity. For simplicity, we discuss the algorithm in terms of the centralized system; however, it can be easily extended to work with the two layered learning algorithms proposed in Section V.

Let $\boldsymbol{\sigma}^n = (\boldsymbol{s}^n, \boldsymbol{a}^n, r^n, \boldsymbol{s}^{n+1})$ represent the ET at stage n, where $\boldsymbol{s}^n = (z^n, q^n, f^n)$, $\boldsymbol{a}^n = (u^n, h^n)$, $r^n = g_2^n - \omega_1 J_1^n - \omega_2 J_2^n$, and $\boldsymbol{s}^{n+1} = (z^{n+1}, q^{n+1}, f^{n+1})$. If the buffer states q^n and q^{n+1} do not satisfy the boundary conditions defined in (13) (i.e. the boundary conditions $q^n = q^{n+1} = 0$ or $q' = Q$), then we can generate *virtual experience tuples* (virtual ETs), which are *statistically equivalent* to the actual ET because of the form of the transition probability and reward functions. This statistical equivalence allows us to perform the Q-learning update step for the virtual ETs using information provided by the actual ET. However, if q^n and q^{n+1} satisfy the

boundary conditions defined in (13), then the statistical information provided by the actual ET is ambiguous and cannot be extended to the non-boundary cases. This is because an ET that satisfies the boundary conditions can be obtained from more than one value of $\tilde{t}$ (i.e. the number of data unit arrivals), while ET's that do not satisfy the boundary conditions correspond to a unique $\tilde{t}$.

We let $\tilde{\boldsymbol{\sigma}}^n = (\tilde{\boldsymbol{s}}, \tilde{\boldsymbol{a}}, \tilde{r}, \tilde{\boldsymbol{s}}') \in \boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$ represent one virtual ET in the set of virtual ETs $\boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$. In order to be statistically equivalent to the actual ET, the virtual ETs in $\boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$ must satisfy the following two conditions:

1. The data unit arrival distribution $p_{\tilde{T}}(\tilde{t} \mid f, z, h)$ defined in (12) must be the same for the virtual ETs as it is for the actual ET. In other words, the virtual operating frequency $\tilde{f}$, the virtual type $\tilde{z}$, and the virtual configuration $\tilde{h}$ must be the same as the actual operating frequency f^n , the actual type z^n , and the actual configuration h^n , respectively. This also means that the virtual costs at the APP and joint OS/HW layers are the same as the actual costs at these layers, i.e. $\tilde{J}_1 = J_1^n$ and $\tilde{J}_2 = J_2^n$.
2. The current virtual buffer state $\tilde{q} \in \mathcal{S}_{\text{APP}}^{(q)}$ and next virtual buffer state $\tilde{q}' \in \mathcal{S}_{\text{APP}}^{(q)}$ must be related to the actual current and next buffer states as follows:

$$\tilde{q}' = \begin{cases} 0, & \tilde{q} + q^{n+1} - q^n < 0 \\ Q, & \tilde{q} + q^{n+1} - q^n > Q \\ \tilde{q} + q^{n+1} - q^n, & \text{otherwise.} \end{cases} \tag{38}$$

Any virtual ET that satisfies the two above conditions can have its reward determined using information embedded in the actual ET. Specifically, from the first condition, we know that the virtual ET's local costs are $\tilde{J}_1 = J_1^n$ and $\tilde{J}_2 = J_2^n$. Then, based on the second condition, we can compute the virtual ET's utility gain as $\tilde{g}_2 = 1 - \left(\frac{\tilde{q} + \lfloor \tilde{t} \rfloor - 1}{Q} \right)^2$, where $\lfloor \tilde{t} \rfloor = q^{n+1} - q^n + 1$ is the number of data unit arrivals that occurred under the actual ET. Finally, the Q-learning update step defined in (20) can be performed on every virtual ET $\tilde{\boldsymbol{\sigma}}^n = (\tilde{\boldsymbol{s}}, \tilde{\boldsymbol{a}}, \tilde{r}, \tilde{\boldsymbol{s}}') \in \boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$ as if it is the actual ET.

Performing the Q-learning update step on every virtual ET in $\boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$ incurs a computational

overhead of approximately $\mathcal{O}\left(\overline{\overline{\boldsymbol{\Sigma}(\boldsymbol{\sigma}^n) \times \boldsymbol{\mathcal{A}}}}\right)$ in time slot n. Unfortunately, it may be impractical to incur such large overheads in every time slot, especially if the data unit granularity is small (e.g. one macroblock). Hence, in our experimental results in Section VII, we show how the learning performance is impacted by updating $\Psi \leq \overline{\overline{\boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)}}$ virtual ETs in each time slot by selecting them randomly and uniformly from the virtual ET set $\boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$.

Table III describes the virtual ET based learning procedure in pseudo-code. Importantly, because the virtual ETs are statistically equivalent to the actual ET, the virtual ET based learning algorithm is *not* an approximation of the Q-learning algorithm; in fact, the proposed algorithm accelerates Q-learning by exploiting our partial knowledge about the structure of the considered problem.

Table III. Accelerated learning using virtual experience tuples.

1.	**Initialize** $Q(\boldsymbol{s},\boldsymbol{a})$ arbitrarily for all $(\boldsymbol{s},\boldsymbol{a}) \in \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}}$;
2.	**Initialize** state $\boldsymbol{s}^0$;
3.	**For** $n = 0,1,\ldots$
4.	Take action $\boldsymbol{a}^n$ using ε-greedy action selection on $Q(\boldsymbol{s}^n,\cdot)$;
5.	Obtain experience tuple $\boldsymbol{\sigma}^n = \left(\boldsymbol{s}^n, \boldsymbol{a}^n, r^n, \boldsymbol{s}^{n+1}\right)$;
6.	**If** NOT($q^n = q^{n+1} = 0$ or $q^{n+1} = Q$) % avoid boundary conditions
7.	% Generate the virtual ET set $\boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$ **For** all $\tilde{q} \in \mathcal{S}_{\text{APP}}^{(q)}$ and all $\tilde{q}'$ that satisfy (38)
8.	$\tilde{\boldsymbol{s}} = \left(\tilde{f}, \tilde{z}, \tilde{q}\right) = (f^n, z^n, \tilde{q})$; % virtual ET state
9.	$\tilde{\boldsymbol{a}} = \left(\tilde{u}, \tilde{h}\right) = (u^n, h^n)$; % virtual ET action
10.	$\tilde{r} = \left[1 - \left(\dfrac{\tilde{q} + \lfloor \tilde{t} \rfloor - 1}{Q}\right)^2\right] - \omega_1 J_1^n - \omega_2 J_2^n$; % virtual ET reward
11.	$\tilde{\boldsymbol{s}}' = \left(\tilde{f}', \tilde{z}', \tilde{q}'\right) = \left(f^{n+1}, z^{n+1}, \tilde{q}'\right)$; % virtual ET next state
12.	$\tilde{\boldsymbol{\sigma}} = \left(\tilde{\boldsymbol{s}}, \tilde{\boldsymbol{a}}, \tilde{r}, \tilde{\boldsymbol{s}}'\right) \in \boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$; % store virtual ET
13.	% Dynamically allocate update steps to the virtual ETs **For** Ψ virtual ETs $\tilde{\boldsymbol{\sigma}} \in \boldsymbol{\Sigma}(\boldsymbol{\sigma}^n)$
14.	$\tilde{\delta} = \left[\tilde{r} + \gamma \max_{\boldsymbol{a}' \in \boldsymbol{\mathcal{A}}} Q(\tilde{\boldsymbol{s}}', \boldsymbol{a}')\right] - Q(\tilde{\boldsymbol{s}}, \tilde{\boldsymbol{a}})$; % TD error for virtual ET $\tilde{\boldsymbol{\sigma}}^n$
15.	$Q(\tilde{\boldsymbol{s}}, \tilde{\boldsymbol{a}}) \leftarrow Q(\tilde{\boldsymbol{s}}, \tilde{\boldsymbol{a}}) + \alpha \tilde{\delta}$; % Q-learning update for virtual ET $\tilde{\boldsymbol{\sigma}}^n$

B. Initializing the Action-Value Function

As we noted before, conventional Q-learning algorithms rely on arbitrary initialization of the action-value function at time $n = 0$ because information about the forms of the reward and transition probability functions is not available a priori. In our setting, however, we are able to

approximate the action-value function. In fact, given our partial knowledge of the reward function, we can determine its upper bound:

$$\begin{aligned} R(\boldsymbol{s},\boldsymbol{a}) &= g(\boldsymbol{s},\boldsymbol{a}) - \sum_{l \in \boldsymbol{\mathcal{L}}} \omega_l J_l(s_l,a_l) \\ &\leq g(\boldsymbol{s},\boldsymbol{a}) \\ &= 1 - \left(\frac{q + \lfloor \tilde{t} \rfloor - 1}{Q} \right)^2 \\ &\leq 1 - \left(\frac{q-1}{Q} \right)^2, \end{aligned} \tag{39}$$

where the second line follows from the first because $\omega_l J_l(s_l,a_l) \geq 0$, for all $l \in \boldsymbol{\mathcal{L}}$, and the forth line follows from the third because $\lfloor \tilde{t} \rfloor \geq 0$. Importantly, the derived upper bound only depends on the state and size of the pre-encoding buffer and does not depend on the system's unknown dynamics; therefore, it can be known offline, prior to run-time.

Using the upper bound on the reward function, we can initialize the action-value function at time $n = 0$ with an approximation of the optimal action-value function $Q^*(\boldsymbol{s},\boldsymbol{a})$. One option is to initialize the action-value function with the optimal action-value function's upper bound. This upper bound occurs when a policy traverses the infinite sequence of buffer states $(q, q-1, q-2, \ldots, 1, 1, \ldots)$ from any initial q, or, equivalently, if the infinite sequence of data unit arrivals $\lfloor \tilde{t} \rfloor$ comprises q 0s followed by infinite 1s. Following such a policy, and using the reward function's upper bound defined in (39), it is easy to see that

$$Q^*(\boldsymbol{s},\boldsymbol{a}) \leq \underbrace{\frac{(\gamma)^q}{1-\gamma}}_{(1,1,\ldots)} + \underbrace{\sum_{\rho=1}^{q} (\gamma)^{q-\rho} \left[1 - \left(\frac{\rho - 1}{Q} \right)^2 \right]}_{(q,q-1,q-2,\ldots,1)}, \tag{40}$$

for all $\boldsymbol{s} = (f,z,q) \in \boldsymbol{\mathcal{S}}$ and all $\boldsymbol{a} = (u,h) \in \boldsymbol{\mathcal{A}}$.

We show experimentally in Appendix B that initializing the action-value function using this upper bound allows Q-learning to perform better than if we initialize $Q(\boldsymbol{s},\boldsymbol{a})$ to 0, for all $(\boldsymbol{s},\boldsymbol{a}) \in \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}}$. However, this upper bound is very optimistic (i.e. it ignores the high costs required to keep the buffer empty), and we will show that better performance is obtained if we approximate the action-value function at time $n = 0$ as

$$Q(\boldsymbol{s},\boldsymbol{a}) = \frac{1}{1-\gamma} \left[1 - \left(\frac{q-1}{Q} \right)^2 \right], \tag{41}$$

for all $\boldsymbol{s} = (f, z, q) \in \boldsymbol{\mathcal{S}}$ and all $\boldsymbol{a} = (u, h) \in \boldsymbol{\mathcal{A}}$. This approximation corresponds to a policy that results in a constant buffer state over time, or, equivalently, results in only one data unit arrival in each time slot. Intuitively, this is a reasonable initial approximation because the true optimal policy will have an average arrival rate of one data unit in each time slot to match the buffer drain rate and avoid buffer overflows.

Suppose we also know the power-frequency function $P(f)$. Since $J_{\text{OS}}(f) = P(f)$ is the immediate cost at the joint OS/HW layer, we can further approximate the initial action-value function as

$$Q(\boldsymbol{s}, \boldsymbol{a}) = \frac{1}{1-\gamma}\left[1 - \left(\frac{q-1}{Q}\right)^2\right] - \omega_{\text{OS}} P(f), \tag{42}$$

for $s_1 = f$, for all $s_2 = (z, q) \in \mathcal{S}_2$, and for all $\boldsymbol{a} = (u, h) \in \boldsymbol{\mathcal{A}}$. In Appendix B, we show experimentally the impact on the learning performance if the power-frequency function is known a priori.

VII. EXPERIMENTS

In this section, we test the performance of the proposed layered learning algorithms using the cross-layer system described in Section III. Table IV details the parameters used in our DMS simulator, which we implemented in Matlab. In our simulations, we use actual video encoder trace data, which we obtained by profiling the H.264 JM Reference Encoder (version 13.2) on a Dell Pentium IV computer. Our traces comprise measurements of the encoded bit-rate (bits/MB), reconstructed distortion (MSE), and encoding complexity (cycles) for each video MB of the *Foreman* sequence (30 Hz, CIF resolution, quantization parameter 24) under three different encoding configurations. The chosen parameters are listed in Table IV. We use a data unit granularity of one macroblock. As in [7], we assume that the power frequency function is of the form $P(f) = \kappa f^{\theta}$, where $\kappa \in \mathbb{R}_+$ and $\theta \in [1,3]$. Since real-time encoding is not possible with the available encoder, we set the data unit arrival rate to $\eta = 44$ DUs/sec, which corresponds to 1/9 frames per second.

Table IV. Simulation parameters.

Layer	Parameter	Value
Application Layer (APP)	Buffer State Set	$\mathcal{S}_{\text{APP}}^{(q)} = \{0, \ldots, Q\}$, $Q = 50$ (DUs)
	Data Unit Type Set	$\mathcal{S}_{\text{APP}}^{(z)} = \{z_1, z_2, z_3\}$ $z_1 = \text{P}$, $z_2 = \text{B}$, $z_3 = \text{I}$

	Parameter Configuration	$\mathcal{A}_{\text{APP}} = \{h_1, h_2, h_3\}$ h_1 : Quarter-pel MV, 8x8 block ME h_2 : Full-pel MV, 8x8 block ME h_3 : Full-pel MV, 16x16 block ME
	APP Cost Weight	$\omega_{\text{APP}} = 22/1875$
	Rate-Distortion Lagrangian	$\lambda_{\text{rd}} = 1/16$
	Data Unit Granularity	1 Macroblock
	Data Unit Arrival Rate	$\eta = 44$ (Data Units/Sec)
Operating System / Hardware Layer (Joint OS/HW)	Power-Frequency Function	$P(f) = \kappa f^{\theta}$ $\kappa \in 1.5 \times 10^{-27}$ and $\theta = 3$.
	Operating Frequency Set	$\mathcal{A}_{\text{OS}} = \{200, 400, 600, 800, 1000\}$ Mhz
	Joint OS/HW Cost Weight	$\omega_{\text{OS}} = 22/125$

A. *Single-layer Learning Results*

In this subsection, we evaluate the learning performance when one layer (say layer l) deploys the local best-response Q-learning algorithm proposed in Section IV.C and the other layer (say layer $-l$) deploys a static policy. For illustration, we assume that the static layer deploys its local optimal policy π_{-l}^* corresponding to the global optimal policy $\pi^* = (\pi_l^*, \pi_{-l}^*)$; hence, the lth layer's best-response policy $\pi_l^*(\pi_{-l}^*)$ is equivalent to π_l^*. Fig. 4 illustrates the optimal policies at the APP layer (π_2^*) and the joint OS/HW layer (π_1^*) for each data unit type when the current operating frequency is $f = 600$ MHz.

Fig. 5 compares the cumulative average reward obtained using single-layer learning to the optimal achievable reward, and Table V shows the corresponding power, rate-distortion costs, utility gain, and buffer overflows, for a simulation of duration $N = 192,000$ time slots (approximately 485 frames drawn from the Foreman sequence, CIF resolution, by repeating the sequence from the beginning after 300 frames). We observe that the global reward obtained when the joint OS/HW layer learns is worse than when the APP layer learns. This is because there are more actions to explore at the joint OS/HW layer (5 compared to 3), and the joint OS/HW layer's policy is more crucial to the system's overall performance than the APP layer's policy. For instance, given the action sets defined in Table IV, the joint OS/HW layer can significantly impact the application's experienced delay (i.e. a factor of 5 change from 200 MHz to 1000 Mhz), while the APP layer cannot (i.e. its actions impact the delay by less than a factor of 2). Consequently, when the APP layer learns in response to the joint OS/HW layer's optimal policy illustrated in Fig. 4(b), the system is initially better off than when the joint OS/HW layer

learns in response to the APP layer's optimal policy illustrated in Fig. 4(a).

We note that, in Fig. 5, the saturation in the performance of the APP layer's best-response learning algorithm is due to the finite exploration probability (i.e. in the ε-greedy action selection procedure $\varepsilon > 0$), which prevents the learning algorithm from ever converging to the optimal policy.

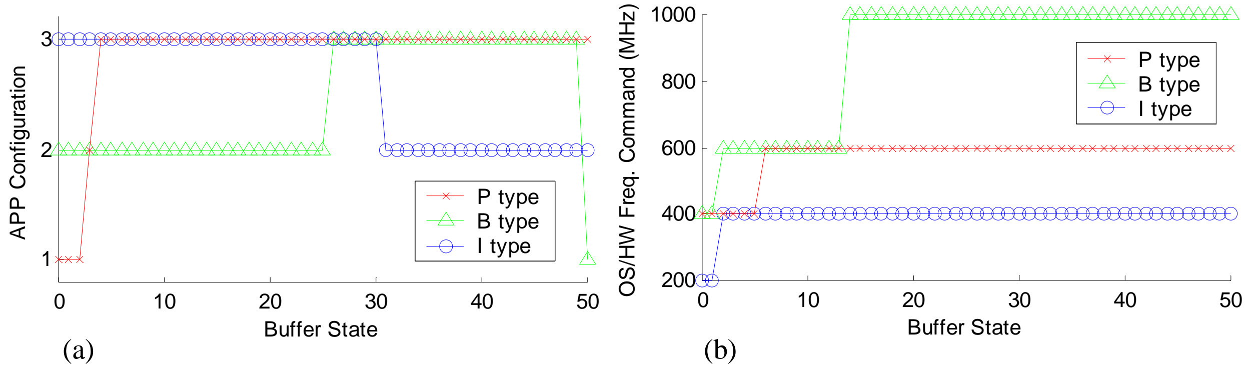

Fig. 4. Optimal policies for each data unit type. (a) The APP layer. (b) The joint OS/HW layer. The current operating frequency is set to be $f = 600$ Mhz.

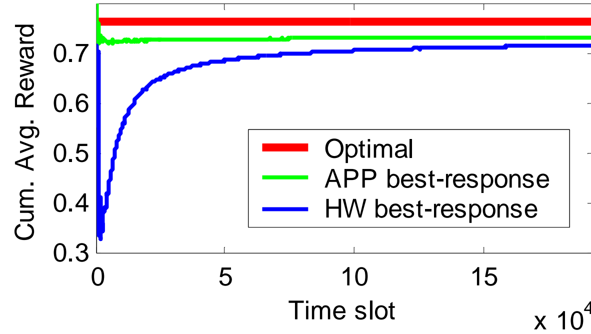

Fig. 5. Cumulative average reward when a single layer learns.

Table V. Single-layer learning performance statistics.

	Best-response APP Layer	Best-response Joint OS/HW Layer	Optimal
Avg. Reward	0.7337	0.7172	0.7631
Avg. Power (W)	0.2835	0.4512	0.2435
Avg. Rate-Distortion	14.93	15.08	14.75
Avg. Utility Gain	0.9588	0.9736	0.9790
No. Overflows	0	61	0

B. Cross-layer Learning Results

In this subsection, we evaluate the learning performance when the layers deploy the coordinated best-response and decentralized Q-learning algorithms proposed in Sections V.A and V.B, respectively. Fig. 6 compares the cumulative average reward obtained using the two decentralized learning algorithms to the performance of the centralized learning algorithm and

the optimal achievable reward; and, Table VI shows the corresponding power, rate-distortion costs, utility gain, and buffer overflows. In these results, the simulation duration is $N = 192,000$ time slots and the coordinated best-response learning algorithm uses $(20,20)$-coordination.

We observe from Fig. 6 that, as expected, the coordinated best-response algorithm performs worse than the centralized and decentralized Q-learning algorithms, because it only allows one layer to learn in each time slot. Meanwhile, the decentralized Q-learning algorithm performs as well as the centralized algorithm *and* adheres to the layered architecture because it allows the layers to act autonomously through decentralized decisions and updates.

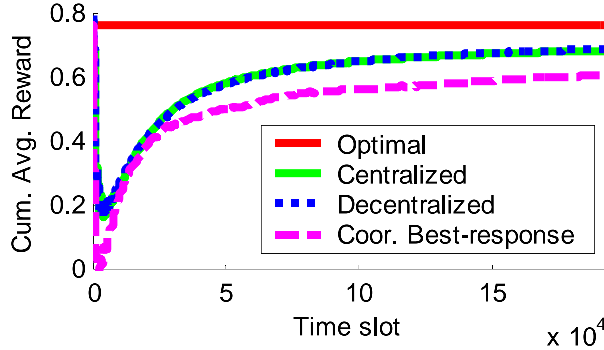

Fig. 6. Cumulative average reward when two layers learn.

Table VI. Cross-layer learning performance statistics.

	Centralized learning	Decentralized learning	Coordinated best-response learning	Optimal
Avg. Reward	0.6829	0.6856	0.6055	0.7631
Avg. Power (W)	0.5441	0.5194	0.5811	0.2435
Avg. Rate-Distortion	15.07	15.08	14.98	14.75
Avg. Utility Gain	0.9555	0.9539	0.8836	0.9790
No. Overflows	339	382	328	0

Table VII shows additional performance statistics for the coordinated best-response learning algorithm using several (N_1, N_2)-Coordination policies. We observe that the choice of N_1 and N_2 significantly impacts the learning performance. In particular, if N_1 and N_2 are too small (e.g. $(N_1, N_2) = (2,2)$), then layer l has trouble improving its performance by learning its best-response policy $\pi_l^*(\boldsymbol{s} \mid \pi_{-l})$ because layer $-l$'s policy π_{-l} changes too frequently. Alternatively, if N_1 and N_2 are too large (e.g. $(N_1, N_2) = (200,200)$), then layer l will spend a long time learning its best-response policy $\pi_l^*(\boldsymbol{s} \mid \pi_{-l})$ to layer $-l$'s *suboptimal* policy π_{-l}, thereby wasting many learning stages without significantly improving the system's performance. Our results show that learning performance is fairly insensitive to changes in N_1 and N_2 between these two extremes.

Table VII. Coordinated best-response learning statistics for different (N_1, N_2) -Coordination policies.

	(N_1, N_2) -Coordinated Best-Response Parameters					
	(2,2)	(5,5)	(10,10)	(20,20)	(50,50)	(200,200)
Avg. Reward	0.5709	0.6020	0.5995	0.6055	0.5952	0.5712
Avg. Power (W)	0.6296	0.6130	0.5991	0.5811	0.5678	0.5587
Avg. Rate-Distortion	14.99	15.04	15.01	14.98	15.06	15.07
Avg. Utility Gain	0.8576	0.8864	0.8810	0.8836	0.8718	0.8501
No. Overflows	697	343	269	328	764	1891

C. Accelerated Learning with Virtual ETs

In this subsection, we evaluate the learning performance when using virtual experience under two different initial conditions. Specifically, we consider the case when $Q(\boldsymbol{s}, \boldsymbol{a})$ is initialized to 0 (case "Zero") and when $Q(\boldsymbol{s}, \boldsymbol{a})$ is initialized with the approximation defined in (41) (case "Approx."). Fig. 7 illustrates the cumulative average reward achieved with a maximum of $\Psi \in \{0, 1, 15, 30, 45\}$ virtual experience updates in each time slot (when $\Psi = 0$ only the actual experience tuple is updated). As expected, increasing the number of updates in each time slot improves learning speed and the system's average performance. What is unexpected is the relative performance of the system under the two different initial conditions. In the upper left plot of Fig. 7 (i.e. no virtual experience), we observe that the "Approx." initial condition drastically outperforms the "zero" initial condition (this is corroborated by the data in appendix B, where we further explore the impact of various initial conditions when there is no virtual experience). However, the other plots in Fig. 7 (i.e. $\Psi \in \{1, 15, 30, 45\}$ virtual updates) show that the "zero" initial condition outperforms the "Approx." initial condition! This is because the "Approx." initial condition approaches the optimal action-value function from *above*, while the "Zero." initial condition approaches it from *below*. To see why this matters, first note that taking the greedy action $\boldsymbol{a}^{n,*} = \arg\max_{\boldsymbol{a}} \{Q^n(\boldsymbol{s}^n, \boldsymbol{a}^n)\}$ induces an action-value update (e.g. (20)), which brings the action-value function's estimate closer to its true value. Now, suppose that this greedy action corresponds to the true optimal action in state $\boldsymbol{s}^n$ (i.e. $\boldsymbol{a}^{n,*} = \pi^*(\boldsymbol{s}^n) = \arg\max_{\boldsymbol{a}} \{Q^*(\boldsymbol{s}^n, \boldsymbol{a}^n)\}$). When approaching the optimal action-value function from *below*, the action-value function update will result in a new action-value $Q^{n+1}(\boldsymbol{s}^n, \boldsymbol{a}^n)$ that is *greater* than the old action-value $Q^n(\boldsymbol{s}^n, \boldsymbol{a}^n)$; hence, the greedy action at time $n+1$ will *still*

be the optimal action. However, when approaching the optimal action-value function from *above*, the action-value function update will result in a new action-value $Q^{n+1}(\boldsymbol{s}^n, \boldsymbol{a}^n)$ that is *less* than the old action-value $Q^n(\boldsymbol{s}^n, \boldsymbol{a}^n)$; consequently, the greedy action at time $n+1$ *may become a sub-optimal action* if $Q^{n+1}(\boldsymbol{s}^n, \boldsymbol{a}^n)$ drops below the second highest action-value in state $\boldsymbol{s}^n$. For this reason, when using virtual experience updates to accelerate the learning process, it is desirable to begin with an initial action-value function that is less than the optimal action-value function for each state-action pair.

We note that, as with the APP layer's best-response learning algorithm, the learning performance with virtual experience saturates due to the finite exploration probability (i.e. $\varepsilon > 0$), which prevents the learning algorithm from converging to the optimal policy.

Table VIII illustrates detailed simulation results that correspond to the experiments in Fig. 7. From this data, it is clear that the initial conditions result in a trade off between delay and power consumption. Specifically, the "Approx." initial condition enables the system to reduce delays and avoid buffer overflows more effectively than the "Zero" initial condition (because the initial conditions mirror the form of the gain function, which regulates delay). Meanwhile, the "Zero" initial condition enables the system to learn the power costs more effectively, resulting in lower power consumption but higher delays and more overflows.

In these experiments, we have assumed that the update complexity is negligible. This would be true if the updates were performed infrequently (e.g. per video frame), however, it is not a valid assumption for the very frequent updates deployed here (i.e. per video macroblock). Hence, performing 15, 30, or 45 updates in each time slot is not reasonable, but performing 1 or 2 virtual ET updates is. Despite this technicality, we have shown the relative performance improvements that can be achieved by updating multiple virtual experience tuples in each time slot. Lastly, we note that the learning performance could be further improved (for the same number of virtual ET updates) by directing the virtual updates to states that are most likely to be visited in the near future, instead of simply randomly updating the virtual experience tuples.

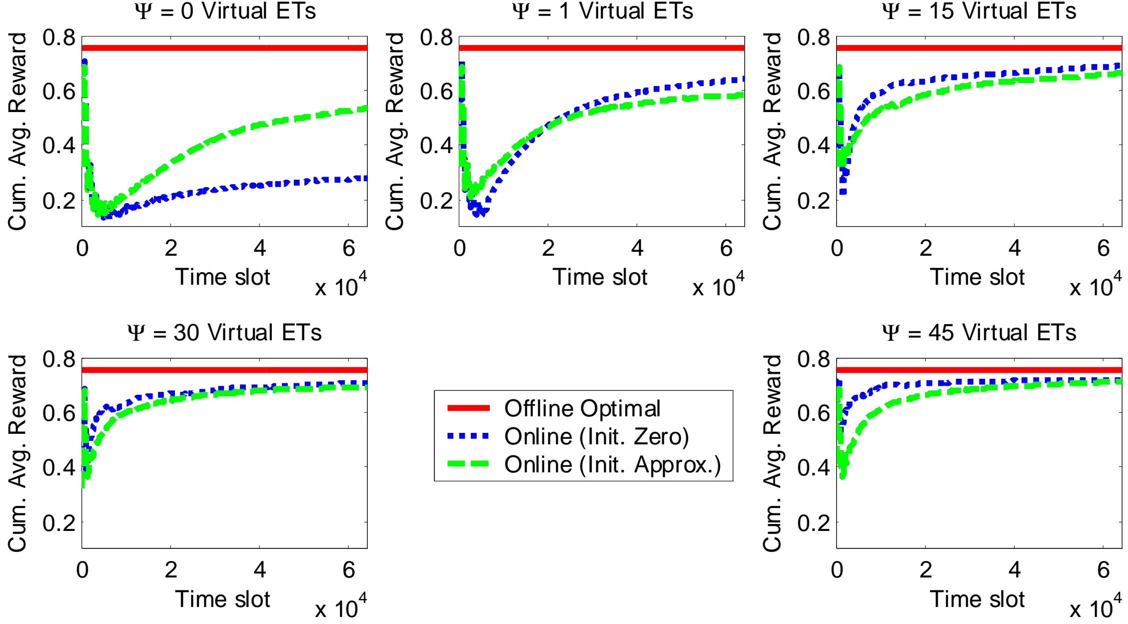

Fig. 7. Cumulative average reward achieved with virtual experience updates under different initial conditions.

Table VIII. Virtual experience learning statistics for different initial conditions. Quantities outside (inside) of parentheses correspond to the "Zero" ("Approx.") initial condition.

	Number of Virtual ET Updates				
	$\Psi = 0$	$\Psi = 1$	$\Psi = 15$	$\Psi = 30$	$\Psi = 45$
Avg. Reward	0.2785 (0.5341)	0.6414 (0.5819)	0.6893 (0.6631)	0.7076 (0.6937)	0.7200 (0.7111)
Avg. Power (W)	0.4212 (0.6836)	0.4164 (0.6576)	0.3817 (0.5250)	0.3432 (0.4380)	0.3108 (0.3898)
Avg. Rate-Distortion	15.00 (15.04)	14.92 (15.07)	15.06 (15.09)	14.93 (15.07)	14.87 (14.93)
Avg. Utility Gain	0.5287 (0.8309)	0.8898 (0.8745)	0.9332 (0.9326)	0.9432 (0.9476)	0.9492 (0.9549)
No. Overflows	1196 (484)	1035 (376)	502 (125)	226 (162)	103 (99)

VIII. Conclusion

We have proposed two novel reinforcement learning algorithms for coordinating the learning processes of the layers in a multi-layer system. The first method coordinates the layers by temporally separating their learning processes so that the learning layer has less "noise" in its experience. The second method uses explicit message exchanges to coordinate the layers. In our experimental results, we verified that the latter layered learning solution achieves the same performance as the centralized solution, which violates the layered architecture. We also exploited our partial knowledge of the system's structure in order to drastically improve the learning performance. Specifically, we found that virtual experience can be exploited to accelerate the rate of learning and that setting good initial conditions greatly improves learning

performance, but that the best choice of initial conditions depends on the deployed learning strategy (e.g. how many virtual experience updates to deploy).

APPENDIX A

In this section of the appendix, we discuss the form of the utility gain function defined in (17). Conventionally, if a multimedia buffer does not overflow (i.e. the buffer constraints are not violated), then there is no penalty. Within the proposed MDP-based framework, where we cannot explicitly impose constraints because the dynamics are not known a priori, the conventional buffer model must be integrated into the system's reward function. Specifically, it can be integrated into the reward through a utility gain function of the form:

$$g_{\mathrm{conv}}(\boldsymbol{s},\boldsymbol{a}) = \begin{cases} 1, & q + \lfloor \tilde{t} \rfloor - 1 \leq Q \\ Q - \left(q + \lfloor \tilde{t} \rfloor - 1\right), & \text{otherwise,} \end{cases}$$

which provides a reward of 1 if the buffer does not overflow, and a penalty proportional to the number of overflows otherwise.

In contrast to the proposed continuous utility gain function defined in (17), $g_{\mathrm{conv}}(\boldsymbol{s},\boldsymbol{a})$ is disjoint. As a result, the optimal foresighted policy obtained using $g_{\mathrm{conv}}(\boldsymbol{s},\boldsymbol{a})$ will initially fill the buffer rapidly in an attempt to minimize rate-distortion and power costs (because it is not penalized for filling the buffer) as illustrated in Fig. 8(a). Subsequently, the policy will attempt to keep the buffer nearly full in order to balance the cost of overflow with the rate-distortion and power costs required to reduce the buffer's occupancy. Unfortunately, with the buffer nearly full, any sudden burst in the complexity of a data unit will immediately overflow the buffer as illustrated in Fig. 8(b). In contrast, the proposed utility gain function is robust against bursts in complexity because it encourages the buffer occupancy to remain low as illustrated in Fig. 8(c,d).

The data in Table IX shows that the proposed utility gain function not only prevents buffer overflows, but it also achieves comparable power consumption as the conventional utility gain function. Thus, we have verified that our choice of utility gain function is good. An added benefit of the proposed utility gain function is that it aids in the learning process. This is because actions are *immediately* rewarded (or penalized) based on how they impact the buffer state.

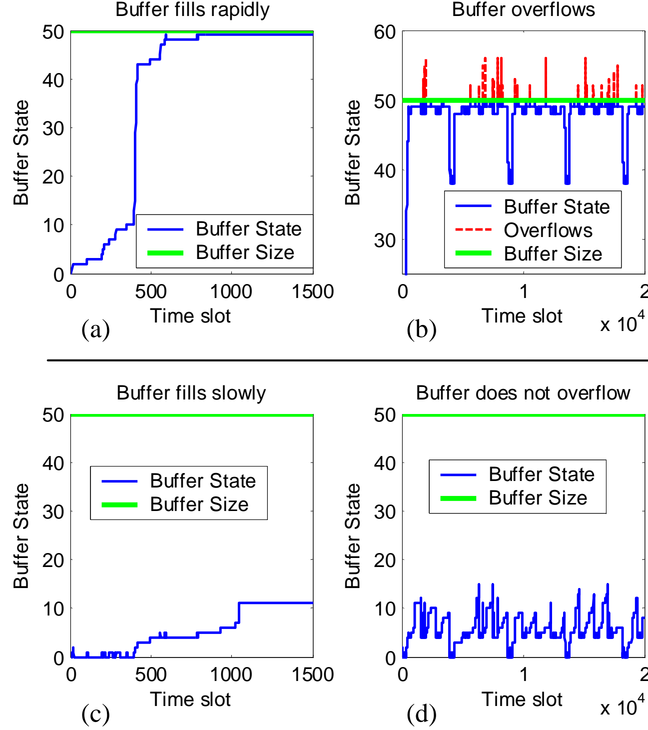

Fig. 8. Buffer evolution in $N = 20,000$ time slot simulation. (a) Conventional utility gain results in the buffer filling rapidly; (b) Conventional utility gain leads to overflows; (c) Proposed utility gain keeps the buffer occupancy low; (d) Proposed utility gain prevents overflows.

Table IX. Performance statistics using the conventional utility gain function and the proposed utility gain function.

	Form of the utility gain	
	Conventional	Proposed
Avg. Reward	0.7785	0.7620
Avg. Power (W)	0.2421	0.2427
Avg. Rate-Distortion	14.95	14.84
Avg. Utility Gain	0.9919	0.9788
No. Overflows	394	0

APPENDIX B

In this section of the appendix, we investigate the impact of the action-value function's initial conditions on the overall learning performance. Specifically, we consider four possible initial conditions: first, $Q(\boldsymbol{s},\boldsymbol{a})$ is initialized to 0 (case "Zero"); second, $Q(\boldsymbol{s},\boldsymbol{a})$ is initialized with its upper bound defined in (40) (case "Upper bound"); third, $Q(\boldsymbol{s},\boldsymbol{a})$ is initialized with the approximation defined in (41) (case "Approx."); and, forth, $Q(\boldsymbol{s},\boldsymbol{a})$ is initialized with the approximation and the known power-frequency costs as defined in (42) (case "Approx. &

power”). Fig. 9 illustrates the cumulative average reward over $N = 128,000$ time slots (macroblock granularity) for all four cases and Table X shows the corresponding average reward, average power, average rate-distortion, average utility gain, and the number of overflows.

From Fig. 9 we observe that setting good initial conditions based on our partial knowledge of the system does indeed improve performance. In all four cases, the reward initially drops because the buffer fills rapidly as actions are explored in each state. In the cases with good initial conditions, the system is able to quickly determine actions to reduce the buffer occupancy, avoid buffer overflows, and improve performance. Interestingly, the optimal learning performance is obtained under the “Approx.” initial conditions, and not the “Approx. & power” initial conditions, which include more information. The “Approx. & power” case performs worse because it initially biases the system toward taking low power (high delay) actions so it takes longer to reduce its buffer occupancy after the buffer initially fills. This observation is corroborated by the data in Table X: The “Approx. & power” case does better in terms of power consumption at the expense of lower utility gain, more buffer overflows, and higher rate-distortion costs.

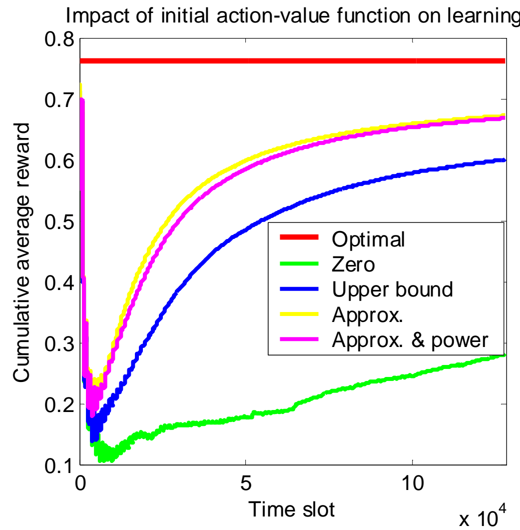

Fig. 9. Learning performance under different action-value function initial conditions.

Table X. Performance statistics under different action-value function initial conditions.

	Initial condition				
	Zero	Upper bound	Approx.	Approx. & power	Optimal
Avg. Reward	0.2808	0.6001	0.6721	0.6681	0.7620
Avg. Power (W)	0.4297	0.5888	0.5523	0.5405	0.2427
Avg. Rate-Distortion	15.15	15.05	15.07	15.13	14.84

Avg. Utility Gain	0.5341	0.8802	0.9461	0.9408	0.9788
No. Overflows	2872	600	325	344	0

REFERENCES

[1] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian, "Integrated power management for video streaming to mobile handheld devices," *Proc. of the 11th ACM international conference on Multimedia*, pp. 582-591, 2003.

[2] R. Cornea, S. Mohapatra, N. Dutt, A. Nicolau, N. Venkatasubramanian, "Managing Cross-Layer Constraints for Interactive Mobile Multimedia," *Proc. of the IEEE Workshop on Constraint-Aware Embedded Software*, 2003.

[3] S. Mohapatra, R. Cornea, H. Oh, K. Lee, M. Kim, N. Dutt, R. Gupta, A. Nicolau, S. Shukla, N. Venkatasubramanian, "A cross-layer approach for power-performance optimization in distributed mobile systems," p. 218a, 19th *IEEE International Parallel and Distributed Processing Symposium*, 2005.

[4] W. Yuan, K. Nahrstedt, S. V. Adve, D. L. Jones, R. H. Kravets, "Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems," *Proc. of SPIE Multimedia Computing and Networking Conference*, vol. 5019, pp. 1-13, Jan. 2003.

[5] W. Yuan, K. Nahrstedt, S. V. Adve, D. L. Jones, R. H. Kravets, "GRACE-1: cross-layer adaptation for multimedia quality and battery energy," *IEEE Trans. on Mobile Computing*, vol. 5, no. 7, pp. 799-815, July 2006.

[6] Z. He, Y. Liang, L. Chen, I. Ahmad, and D. Wu, "Power-rate-distortion analysis for wireless video communication under energy constraints," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 645-658, May 2005.

[7] Z. He, W. Cheng, X. Chen, "Energy minimization of portable video communication devices based on power-rate-distortion optimization," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 18, no. 5, May 2008.

[8] D. G. Sachs, S. Adve, D. L. Jones, "Cross-layer adaptive video coding to reduce energy on general-purpose processors," in *Proc. International Conference on Image Processing*, vol. 3, pp. III-109-112 vol. 2, Sept. 2003.

[9] N. Mastronarde and M. van der Schaar, "Towards a general framework for cross-layer decision making in multimedia systems," *IEEE Trans. on Circuits and Systems for Video Technology,* to appear.

[10] S. Irani, G. Singh, S. K. Shukla, R. K. Gupta, "An overview of the competitive and adversarial approaches to designing dynamic power management strategies," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 13, no. 12, pp. 1349-1361, Dec. 2005.

[11] L. Benini, A. Bogliolo, G. A. Paleologo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Trans. on computer-aided design of integrated circuits*, vol. 18, no. 6, June 1999.

[12] E.-Y. Chung, L. Benini, A. Bogliolo, Y.-H. Lu, and G. De Micheli, "Dynamic power management for nonstationary service requests," *IEEE Trans. on Computers*, vol. 51, no. 11, Nov. 2002.

[13] Z. Ren, B. H. Krogh, R. Marculescu, "Hierarchical adaptive dynamic power management," *IEEE Trans. on Computers*, vol. 54, no. 4, Apr. 2005.

[14] R. S. Sutton, and A. G. Barto, "Reinforcement learning: an introduction," Cambridge, MA:MIT press, 1998.

[15] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: a survey,” *Journal of Artificial Intelligence Research 4*, pp. 237-285, May 2005.
[16] C. J. C. H. Watkins and P. Dayan, “Technical Note: Q-learning,” *Machine Learning*, vol. 8, no. 3-4, pp. 279-292, May 1992.
[17] S. Russell and A. L. Zimdars, “Q-decomposition for reinforcement learning agents,” in *Proc. of the International Conference on Machine Learning*, pp. 656-663, 2003.
[18] J. O. Kephart, H. Chan, R. Das, D. W. Levine, G. Tesauro, F. Rawson, and C. Lefurgy, “Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs,” *Proc. of the 4th International Conference on Autonomic Computer*, 2007.
[19] Omer F. Rana and Jeffrey O. Kephart, "Building Effective Multivendor Autonomic Computing Systems," *IEEE Distributed Systems Online*, vol. 7, no. 9, 2006, art. no. 0609-o9003.
[20] D. S. Turaga and T. Chen, “Hierarchical modeling of variable bit rate video sources,” Packet Video Workshop, May 2001.
[21] A. Ortega, K. Ramchandran, M. Vetterli, “Optimal trellis-based buffered compression and fast approximations,” *IEEE Trans. on Image Processing*, vol. 3, no. 1, pp. 26-40, Jan. 1994.
[22] E. Akyol and and M. van der Schaar, "Complexity Model Based Proactive Dynamic Voltage Scaling for Video Decoding Systems," *IEEE Trans. Multimedia*, vol. 9, no. 7, pp. 1475-1492, Nov. 2007.
[23] S. P. Sanner, “First-order decision-theoretic planning in structured relational environments,” Ph.D. Thesis, University of Toronto, 2008.
[24] N. Nahrstedt, D. Xu, D. Wichadakul, and B. Li QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments, IEEE Communications Magazine, 2001.
[25] S. Mohapatra and N. Venkatasubramanian, “PARM: Power-aware reconfigurable middleware,” Proc. 23rd Internat. Conf. on Distributed Computing Systems, 2003.
[26] J. Nieh, M.S. Lam, “The design, implementation and evaluation of SMART: a scheduler for multimedia applications,” *Proc. of the Sixteenth ACM Symposium on Operating Systems Principles*, pp. 184-197, Oct. 1997.
[27] P. Goyal, X. Guo, H.M. Vin, “A Hierarchical CPU Scheduler for Multimedia Operating Systems,” Usenix 2nd Symposium on OS Design and Implementation, pp. 107-122, 1996.
[28] P. A. Dinda, and D. R. O’Hallaron, “An evaluation of linear models for host load prediction,” Proc. IEEE Internat. Sympos. High Perf. Distrib. Comput., pp. 87-96, Aug. 1999.
[29] Y. Andreopoulos and M. van der Schaar, "Adaptive Linear Prediction for Resource Estimation of Video Decoding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 17, no. 6, pp. 751-764, June 2007.
[30] Y. Shoham and K. Leyton-Brown, *Multi-agent Systems: Algorithmic, Game Theoretic and Logical Foundations*, Cambridge University Press, 2008.