2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) | 978-1-6654-3860-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/MCSOC51149.2021.00040

2QoSM: A Q-Learner QoS Manager for Application-Guided Power-Aware Systems

Michael J. Giardino and Daniel Schwyn
Systems Group - Department of Computer Science
ETH Zürich
Zürich, Switzerland
{michael.giardino,daniel.schwyn}@inf.ethz.ch

Bonnie Ferri
School of Electrical Engineering
Georgia Institute of Technology
Atlanta, United States
bonnie.ferri@gatech.edu

Aldo Ferri
School of Mechanical Engineering
Georgia Institute of Technology
Atlanta, United States
al.ferri@me.gatech.edu

***Abstract*—This paper describes the design and performance of Q-learning-based quality-of-service manager (2QoSM) for compute-aware applications (CAAs) as part of platform-agnostic resource management framework. CAAs and hardware are able to share metrics of performance with the 2QoSM and the 2QoSM can attempt to reconfigure CAAs and hardware to meet performance targets. This enables many co-design benefits while allowing for policy and platform portability. The use of Q-Learning allows online generation of the power management policy without requiring details about system state or actions, and can meet different goals including error, power minimization, or a combination of both. 2QoSM, evaluated using an embedded MCSoC controlling a mobile robot, reduces power compared to the Linux on-demand governor by 38.7-42.6% and a situation-aware governor by 4.0-10.2%. An error-minimization policy obtained a reduction in path-following error of 4.6-8.9%.**

I. Introduction

Whether dealing with battery-powered mobile robots or data centers, computing devices remain power constrained. These constraints manifest in thermal limits, battery capacity, economic cost, or physical power delivery. What the term *power management* entails depends on specific devices, but fundamentally computer systems control power use by reducing the performance of devices in the system.

In the case of CPUs, this usually involves dynamic voltage and frequency scaling (DVFS) [1] and power gating [2] whereas DRAM may use reduced refresh rates [3] and hard drives may spin down [4]. These techniques reduce performance, but if a device is underutilized or idle, the performance reduction in exchange for power is sensible. From the software's perspective, real-time requirements, interactivity, or boundedness (i.e. memory- vs compute-bound) can complicate power management. Modern and emerging hardware such as heterogeneous compute (e.g. big.LITTLE) and memory (DRAM + NVM) and peripherals with large power budgets (e.g. disks or coprocessors) further add complexity.

As discussed in Section II, there are many techniques of software-controlled power management. In one extreme, you have one-size-fits-all algorithms of general purpose operating systems (e.g. Linux's on-demand governor) which work well across systems, but due to this generality, remain greatly sub-optimal. At the other end, there is co-design, wherein policies and applications are tailored to specific hardware and have direct control of management. Co-design can provide strong guarantees of correctness and near-optimal performance/power consumption, but the tight coupling requires significant engineering and eliminates cross-platform portability. With the huge number of discrete platforms, especially among single-board computers (SBCs) and systems-on-chips (SoCs), the lack of portability is problematic.

An ideal system would allow interaction between application and hardware but without the tight integration that hurts portability. Our work demonstrates such a system by allowing application state, performance, and guidance as well as hardware state and control to be abstracted and managed through a power controller. This controller need not understand the specifics of state or controls, only that its actions result in state changes, and to determine which decisions provide the best outcomes from a given state. Moreover, by providing a communications channel between application and hardware, we no longer have to rely on basic CPU performance metrics, but can instead use both quantitative and qualitative application and whole-system performance. Application performance metrics are especially important in physical systems (e.g. mobile robots) because we are concerned with both the speed of execution and the *quality of execution*. Since many of these algorithms are non-deterministic, computational system performance affects the *quality* of the solution.

The contributions presented in this paper are as follows:

- an abstraction of physical system performance (communication and path error), application state, and power consumption into a unitless state vector
- a C-based Q-learner-based Quality-of-Service Manager (2QoSM) that takes the state vector, and learns policies for system power management based upon tunable reward functions
- an evaluation of the 2QoSM on a MCSoC at the center of a power-constrained mobile robot showing significant improvement over both standard Linux power management and a state-of-the-art application-guided governor
- a demonstration of the utility of application, hardware, and policy agnostic middleware for embedded power management

978-1-6654-3860-5/21/$31.00 ©2021 IEEE
DOI 10.1109/MCSoC51149.2021.00040

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on June 20,2023 at 23:12:54 UTC from IEEE Xplore. Restrictions apply.

II. Related Work

The power consumed by computer system components vary tremendously depending on the platform. Early PC power consumption looked surprisingly different. A computer in the mid-1990s might have nearly three quarters of its power budget consumed by the monitor, 20% from the hard drive, and less than a watt consumed by CPU and memory combined [4]. A modern CPU, by comparison, consumes much more power, anywhere from less than 5 W for a laptop processor to 400 W for a server processor (5-800x) [5]. Memory in enterprise and HPC systems can consume up to 40% of total power budget [6]. For mobile devices, the energy consumed by the OLED/LCD display may consume 50% [7] or more [8] while embedded IoT devices expend the vast majority of their energy budgets on networking [9]. Clearly, the power needs of these different systems vary widely and require a closer examination of their power management schemes.

One of the most important technological advances in computer power management was dynamic voltage and frequency scaling (DVFS). While CPU power consumption is determined by many factors [10], voltage and frequency are the only runtime controllable variables allowing for a simplification of CPU power [1] as:

$$P_{CPU} \propto V^2 \cdot f \tag{1}$$

where the power consumed by the CPU P_{CPU} is directly proportional to the square of the voltage V times the clock frequency f. Thus, runtime adjustment of these variables is the dominant form of CPU power management. Many modern CPUs have fine-grained on-die control of voltage and frequency, however the vast majority of processors, especially SoCs, rely on external software control for DVFS state management.

The most common method of software-controlled DVFS (e.g. Linux on-demand governor [11]) is to set a target system load, and adjust the P-States (discrete voltage/frequency pairs) to reach it. More sophisticated strategies for controlling DVFS include scheduling periodic workloads [12], integrated runtime systems such as CPU MISER [13], CoScale [14], or Intel's RAPL system based upon energy quotas [15].

Modern processors can quickly power down regions of the CPU (power-gating) allowing for an alternative power-management paradigm: race-to-idle [16] or race-to-halt [17]. Instead of optimizing P-states, race-to-halt uses the highest performance state to complete the work, and immediately attempts to power down. DVFS states, especially when attempting race-to-halt, rely on the assumption that tasks are *compute bound* (versus *memory bound*). A computationally-bound process is only limited by the number of CPU instructions retired and directly benefits from increasing CPU frequency. However, if the process is memory bound, increasing the CPU performance will use more power without significantly speeding up execution [18]. Mixed workloads are more of a challenge because they may benefit from a combination of policies as the program changes execution regions. An advantage to using a machine-learning is that if the best paradigm for a given workload/system combination is race-to-halt, the policy will converge to it, while still being able to adjust for more memory-bound or mixed workloads.

The combination of increased computational performance and energy consumption motivates the use of CPU cycles for more efficient, predictive power management and scheduling. These systems go by various names but we will refer to them generally as "quality-of-service managers" (QoSM) to include the broad spectrum of hardware and software targets, constraints, and goals. Two closest related areas of modern power management related to our work are QoSM *middleware* and machine learning-based power managers.

The use of middleware, a software abstraction layer between application and operating system, for power and performance management is well-studied. Li *et al.* developed performance-aware middleware for distributed video systems in an attempt to balance the objectives of hardware and application [19]. Shortly after, ControlWare provided QoS management in distributed real-time systems that lies between hard and probabilistic performance guarantees [20]. Hoffman *et al.* demonstrate the need for coordination in optimizing for accuracy and power and propose CoAdapt, a dynamic coordinated controller for optimizing performance, accuracy and power [21]. We also attempt to balance accuracy (error) and power consumption by adjusting hardware and application algorithm but we diverge in target application/hardware (embedded v. enterprise) and choice of QoSM (feedback control v. machine learning). POET, a C-based framework, minimizes energy consumption while still meeting soft real-time constraints [22]. Imes *et al.* expanded upon their work on POET to create Bard which allows for runtime switching between power and performance constraints [23]. POET/Bard are closely related both in their framework architecture and the ODROID-XU3 experimental platform. Our approach differs in that Bard requires a pre-computed set of frequencies and associated speedups/power usage whereas our learner simply has knobs to turn and metrics to read. This benefits mixed or memory-intensive workloads when a calculated speedup associated with a P-state is only a best-case scenario for a compute-bound workload.

Given the demonstrated improvement of the application-guided situation-aware governor over Linux's on-demand governor, we designed the 2QoSM described in this work as as a drop-in replacement [24]. This work makes use of the existing framework and experimental platform to demonstrate both the ease of replacing the governor/QoSM, as well as the improvements of Q-learning over a custom, context-aware governor. Additional differentiation between the two implementations is described in Section III.

The framing of power management as a machine learning problem is discussed in more detail in Section III-B. However, there is a spate of recent machine learning based power managers, especially those using reinforcement learning to determine the optimal power state of a CPU [25]. Martinez and Ipek [26] examine the ability of machine learning to make low-level power management decisions including

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on June 20,2023 at 23:12:54 UTC from IEEE Xplore. Restrictions apply.

DRAM scheduling and multiresource allocation. Ye and Xu use Q-Learning to optimize idle periods to reduce power consumption in simulations of synthetic benchmarks [27]. Their Q-Learning-based quality-of-service manager (2QoSM) differs in that their goal is to reduce transitions, while our 2QoSM is only concerned with the measured power consumption. If transitioning too often is increasing power use through leakage current, this is reflected in power consumption and 2QoSM can adjust the policy accordingly. Shen *et al.* [28] use a Q-Learner to pick DVFS states by constraining the performance and temperature while minimizing the total energy. Similarly, Ge *et al.* use CPU metrics, user-constraints, and processor temperature for determining state and reward [29]. Many of the design decisions they made are similar to the ones made in this paper, but our work has some notable differences. The most important difference is that our reward and state are not based upon CPU utilization (i.e. IPS and CPU intensiveness) but instead the measured performance of the application, which is a more represenatitive metric of total system performance than CPU load. Additionally, user-defined constraints can be overconstrained and the learner cannot find a policy. Das *et al.* use a Q-Learner to allocate threads and set CPU frequency to obtain a given performance target [30]. Their work is focused on thermal limits and performance targets are time-based deadlines while our application's execution is open-ended thus we have no *a priori* knowledge of execution time. Therefore we must use runtime metrics to determine application performance. Deep Q-Learning (DQL) was used on a similar big.LITTLE single-board computer to obtain near-optimal performance per watt [31]. While we believe DQL is a viable approach to power management, it does have a few drawbacks that our work does not. Their system requires the development of an Oracle which demands a significant amount of offline training and benchmarking, which for a non-deterministic physical system may be entirely infeasible.

III. Architecture and Quality of Service Manager (QoSM)

Section III-A will give a brief overview of the design and motivation of the architecture, and focus on specifics that differ from previous work. A detailed discussion of Q-Learning, its place in modern power management, and details of implementation are discussed in Section III-B.

A. Software Architecture

In order to meet our requirements of abstraction and controlability, we use a layered architecture, shown in Figure 1, made of three primary components: *compute-aware applications*, *hardware abstraction layer*, and a *quality of service manager*. As discussed in Section II, the framework is based upon that developed in [24] with some notable differences as discussed below.

We define Compute-Aware Applications (CAAs) as applications which have meet the two requirements of controlability: the ability to dynamically change algorithmic profiles, and provide a metric of progress or performance. Multiple CAAs

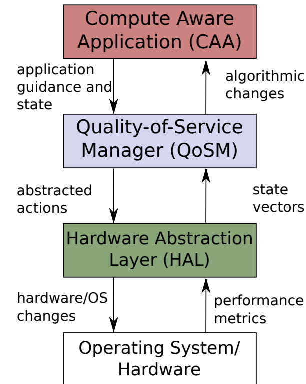

Fig. 1. A high level view of the software framework architecture.

can be composed and individual results passed to the QoSM, which continually monitors the CAA's state. The CAA state consists of both metrics (e.g. SNR, latency, control system error signals), and application guidance to the hardware. It is important to note that, especially in relation to the situation-aware governor [24], QoSM does not have to know what the metric or guidance *means*, it simply must be able to integrate it into a unitless state vector. The QoSM uses a policy or set of policies, in this case a Q-Learner, to take an action. Again, this action does not have to be understood by the QoSM, it simply must be able to observe the outcome. One can think of these actions as a dial without a label that the learner turns and observes the outcome. These actions are given to the hardware abstraction layer (HAL), which converts them into low-level, platform specific hardware/operating system changes.

Moving back up the software stack, the HAL receives metrics from the OS (e.g. performance counters, temperature, or power measurements) which are then packed into a state vector and passed to the QoSM. In the same way the QoSM passes actions to the HAL, it also may suggest algorithmic changes to the CAA. For example, if the QoSM finds itself with excess capacity, the QoSM can inform the CAA to use higher performance algorithms (e.g. deeper searches). When capacity is limited, the QoSM can notify CAAs, allowing applications the ability to degrade gracefully. This coordination between application and hardware enables the development of more intelligent and tailored policies.

B. Q-Learner Quality of Service Manager (2QoSM)

As discussed above in Section II, reinforcement learning is a natural choice for management of both CPU P-states and application profiles. Q-Learning is a technique for reinforcement learning that does not require any knowledge of an underlying model [32]. Instead of creating a model, Q-Learning estimates a real-valued function Q of states and actions where $Q(s, a)$ is the expected discounted sum of future rewards for performing action a in state s [33]. The values of Q are stored in a matrix of dimensions $s \times a$. Q is calculated using the function

$$Q'(s_t, a_t) = (1-\alpha) \cdot Q(s, a) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a_t)) \quad (2)$$

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on June 20,2023 at 23:12:54 UTC from IEEE Xplore. Restrictions apply.

where $Q'(s_t, a_t)$ is the updated value in the Q-matrix, α is a learning rate, r_t is the instantaneous reward, γ is the discount factor, and $\max_a Q(s_{t+1}, a_t)$ is an estimate of optimal future value. The learning rate $\alpha \in [0, 1]$ is a parameter used to balance the incorporation of new information with previously calculated values. With a learning rate closer to 1, the update of Q is more affected by the new reward, whereas a smaller α weights existing Q values higher. The discount factor γ determines the weighting of future versus present rewards with a large γ favoring future rewards and a small γ maximizing immediate rewards. $\gamma = 0$ creates an algorithm that only maximizes instantaneous rewards, while $\gamma \geq 1$ will not converge and $Q \rightarrow \infty$. The optimal future value is the best possible Q available in the next iteration after taking best action a. The algorithm for updating the Q-matrix every step is shown as Algorithm 2.

1: **if** t = 0 **then**
2: Initialize Q-matrix $\forall a, s, Q(s, a) = 0$
3: **end if**
4: **while** $s_{t+1} \neq endstate$ **do**
5: Observe current state s_t
6: Find $\max_a Q(s_t, a)$
7: Take action a based upon selection policy
8: Calculate instantaneous reward r
9: Update Q based upon Equation 2
10: **end while**

Fig. 2. Q-Learning Update

The Q-matrix of size $S \times A$, where S is the number of states and A is the number of actions is allocated at startup, and, in our implementation, initialized to zero. It is possible to initialize the matrix to random values [34] or to use more complex methods of pre-populating the Q-Matrix for faster convergence [35] The learner observes its current state s_t, and looks up row s_t in its Q-table. It examines the values in row $Q(s_t)$ and takes the action based upon its selection policy, most often the action with the highest $q_{s_t,a}$. During training, it's good practice to take a random action with probability ρ to better explore the action-reward space. Once the action is taken, the reward r is measured, and the q-value $Q_{new}(s_t, a_t)$ is updated based upon Equation 2.

In the problems often tackled by Q-Learning, the algorithm ends when a specific goal state is reached, the system is restarted, and learning continues from the beginning. However, we are using Q-learning for non-episodic tasks which have no terminal state. As long as the system finds itself in previously encountered states, the learner can develop its Q-function to obtain maximum future rewards. As the system moves through states taking actions, the difference between successive updates to a given $Q(s, a)$ begins to decrease, and the learner converges. In infinite time the Q-Learner converges to an optimal policy, however the parameters of γ, α and ρ can lead to different trained learners in finite time [36].

IV. Experimental Results

The 2QoSM was tested on an autonomous robot which was chosen for a few reasons. For one, it has a limited power budget and its power consumption is dominated by two similarly power-hungry components: a fairly powerful MCSoC and motors. This is a useful characteristic for testing because a reduction in CPU energy consumption directly allows for a longer autonomous runtime. Moreover, it there is a direct relationship between motor energy consumed on a less optimal path and the energy saved by reduced CPU usage. Second, the algorithm for route planning is computationally intensive, and depending on the computational cycles allocated to it, gives progressively better results. Third, there is a metric of system performance, namely path error. Section IV-A details the test platform, Section IV-B elaborates on the software implementation of the Q-Learner-based Quality-of-Service Manager (2QoSM), Section IV-C examines the training and convergence, and Sections IV-D and IV-E discuss the overall performance results of the power manager.

A. Experimental Platform Details

The robot is a heavily-modified DF Robot Cherokey 4WD [37] using a custom stacked heterogeneous architecture. The computational system consists of an Arduino ROMEO BLE ATMega328/P [38] controlling the low-level motor controllers and sensors, and a ODROID XU4 [39] running networking, mapping, route planning, and trajectory planning.

The ODROID XU4 is a powerful multicore SoC (MCSoC) single-board computer (SBC) and has been used as a test platform in much previous work in intelligent power management [22]–[24], [31], [40], [41]. This ARM big.LITTLE heterogeneous multiprocessing architecture (HMP) provides thread scheduling between higher-performance higher-power cores (Cortex-A15) and lower-performance lower-power cores (Cortex-A7) and per-cluster DVFS. The XU4 is running Ubuntu Linux with kernel v4.14.5-92. The robot traverses an 12 m x 12 m slalom course that divides this area into 1,440,000 1 cm squares. The robot uses Anytime Dynamic A* [42] to chart a path around obstacles it observes. A feature of ADA* that makes it especially suitable for use as a CAA is that it iteratively and monotonically improves the quality of the path during execution. A single iteration is enough for a correct path through the obstacles, however, given more resources (execution time), it is able to find a more efficient path. When new obstacles are observed, the planner activates and recalculates the path The trajectory planning thread determines the maneuvers that are needed to follow the path based upon the information obtained from odometry calculations, and then passes these commands to the ATMega328.

B. Quality of Service Manager Implementation

Instead of using an existing framework for Q-Learning, we opted for implementing our Q-Learner in C using GNU Scientific Library [43] for a few reasons. First, most machine learning libraries are written in Python which is unsuitable for systems programming. Second, while there exist major

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on June 20,2023 at 23:12:54 UTC from IEEE Xplore. Restrictions apply.

C++ implementations of machine learning libraries, this still requires moving from an entirely C-based approach to C++ and introduces additional complexity. Finally, the Q-Learning algorithm is relatively simple and was able to be implemented using only the existing GSL library.

TABLE I
COMPOSITION OF STATE VECTOR

metric	no. of levels	cumulative states	size (B)
error	10	10	400 B
power consumption	10	100	4000 B
replanning mode	2	200	8000 B
checksum error	4	400	16000 B
map updates	2	1600	64000 B

The Q-Learner runs as a separate thread from the navigation algorithm. In this experiment, the 2QoSM operates on a 10 ms sample period. When the timer expires, the 2QoSM tests the conditional at line 4 of Algorithm 2, and if the robot has not reached its goal, it begins its update. First, it reads the available metrics, discretizes them, and packages them to states. Several different state combinations were assessed during the course of this research, but the data presented in this paper use the state variables shown in Table I. As can be seen from the third column of Table I, the number of states increases multiplicatively with the number of levels of each additional state variable. Each entry of the Q-matrix is of type `double` which in ARMv7 is 64 bits (8 B) and there are five entries, one per action, per row. While each individual row in the Q-matrix isn't large (40 B), the rapid increase in matrix size makes selecting useful state variables and their precision quite important. That said, even with 1600 states, the total size of the matrix is only 64 KB. One could easily halve the size of the matrix by replacing `double` with 32 b `float`. After determining its current state, the learner finds the row in its Q-table associated with the state, and determines the best action based upon the current policy. The default policy is to choose the action with the highest future reward with probability $\rho = 0.9$ and select a random action with probability 0.1. After taking its action, it observes the reward obtained by the action, and updates the associated $Q(s_t, a_t)$. Finally, the timer is reset and the 2QoSM thread goes back to sleep.

C. Training and Convergence

The Q-Learner was trained by having the robot navigate to randomly defined coordinates in an environment with obstacles. Each tested reward function was trained for 600 s before running the experiments shown in Section IV. Because Q-Learning continues to improve its policy online, every following run would have a further-trained learner. In our results, we did not see any indication that further runs gave a noticeable improvement in performance even though the policy would continue to converge. However, to better compare the results, after 600 s of training we froze the Q-matrix and reused it to start each run.

The variables used for describing the reward functions are shown in Table II.

TABLE II
SYMBOLS USED IN REWARD FUNCTIONS

Symbol	Description
t	Time of sample
P_t	Instantaneous Power at time t
P_{max}	Maximum measured power
E_t	Error measured at time t
E_{max}	Maximum measured error
r	ADA* replanning active (0, 1)
$\sum r$	Accumulating sum of replanning
$\bar{P}_n$	n-Sample moving Average of Power

TABLE III
REWARD FUNCTIONS TESTED IN 2QOSM

Reward Description	Reward Function
Power Only	$1 - \frac{P_t}{P_{max}}$
Error Only	$1 - \frac{E_t}{E_{max}}$
Power + Error	$2 - \left(\frac{P_t}{P_{max}} + \frac{E_t}{E_{max}}\right)$
Power + Error + Replan Flag	$3 - \left(\frac{P_t}{P_{max}} + \frac{E_t}{E_{max}} + r\right)$
Power + Error + Accumulating Replan Flag	$3 - \left(\frac{P_t}{P_{max}} + \frac{E_t}{E_{max}} + \sum r\right)$
Weighted 1:10 power:error	$1 - \left(0.1 \cdot \frac{P_t}{P_{max}} + 0.9 \cdot \frac{E_t}{E_{max}}\right)$
Error + 10-sample Moving Average of Power	$2 - \left(E_t + \bar{P}_{10}\right)$

D. Metrics

To evaluate the performance of different reward functions defined in Table III), we've compared each trained learner to the default Linux on-demand governor [11] and the situation-aware governor [24]. The left part of Figure 3 shows metrics of path error, average power, and the time required to navigate the course. This is average data collected from 10 runs per reward/governor. The Q-Learner updates its Q-matrix during the run, but to maintain a constant baseline, reverts to the original trained matrix before starting successive runs.

The functionality of the Q-Learner is demonstrated by comparing the power-only and error-only reward functions. When the reward function optimizes for power, it has the lowest power consumption of any other configuration, however it also has a significantly higher error. On the other hand, the error-only reward function manages to obtain the minimum error but at the cost of highest power consumption of all governors other than the default Linux on-demand. The runtime remains approximately the same for all governors, thus the reduction in average CPU power correspond to total energy savings.

The lowest power consumption is shown by the power-only reward function, reducing power consumption by 2.98 W

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on June 20,2023 at 23:12:54 UTC from IEEE Xplore. Restrictions apply.

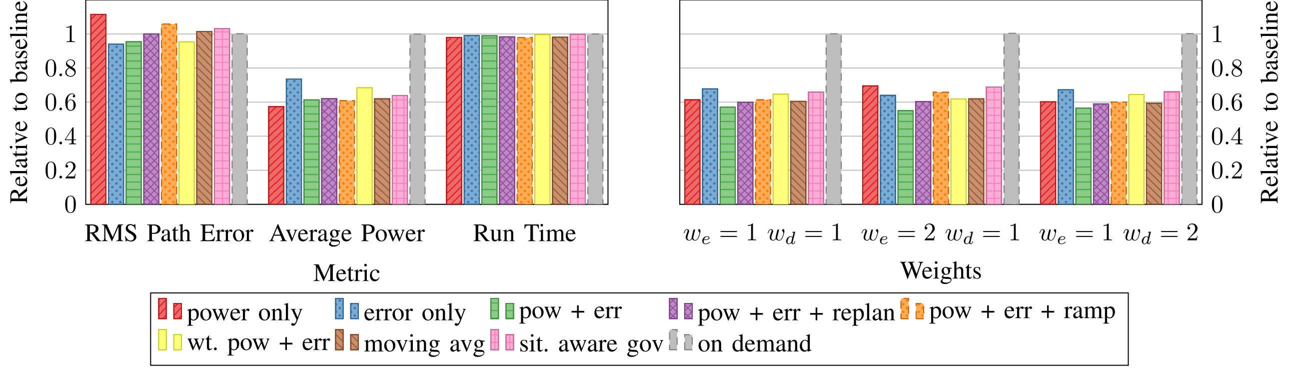

Fig. 3. The collected metrics of power, performance, and runtime on the left, Energy-Error Delay Product [24] of the reward functions with different weights on the right

(42.6%) compared to the the default Linux governor and 0.457 W (10.2%) compared to the situation-aware governor. The lowest RMS error is found by using an error-only reward function, obtaining a 6.1% improvement over the on-demand governor and 8.9% over the situation-aware governor. Using the sum of power and error reward function reduces power compared to the Linux on-demand governor by 2.703 W (38.7%) and the situation-aware governor by 0.18 W (4.0%). The error is also improved over the on-demand and situation-aware governors by 4.6% and 7.49% respectively.

Because the improvement of physical system application performance lies not only in the speed of completing the task, but the *quality* of the solution, instead of measuring the energy-delay product (EDP), we use the energy-error delay product (EEDP) [24]

$$\text{EEDP} = E \cdot \epsilon^{w_e} \cdot T^{w_d} \tag{3}$$

where E and T are normalized energy and runtime, ϵ is the normalized system error, w_ϵ is the weight placed on the error, and w_d is the weight placed on delay. With the inclusion of error into the metrics, we can evaluate different reward functions more clearly, differentiating their performance based upon the importance we place on energy, error, and delay. These results are shown in Figure 3 on the right. If we weight error and delay the same, the power-only reward function still outperforms all others. However, if we put extra weight on either error or delay, the sum of power and error as a reward function shows the greatest performance. We believe that the addition of EEDP as a metric for evaluating power management decisions allows for researchers to not only better differentiate between the aggregate behavior of different algorithms, but to select the system that best meets the targeted goals of the designer.

E. Time Series Evaluation

For a better understanding of what is happening during the experiments, we present time-series data, collected at every update of the algorithm (10 ms) during the course of a run. While the previous data presented in Section IV-D are averaged across 10 runs, the data below represent a single representative traversal of the course. Due to space constraints, we will only present a subset of the reward functions. The top figure contains the measured current in blue (a proxy for power since the voltage provided by regulators is constant), the measured error in green (deviation from the ideal path), and whether replanning is active or not in red/black. The bottom figure shows the instantaneous reward during the run.

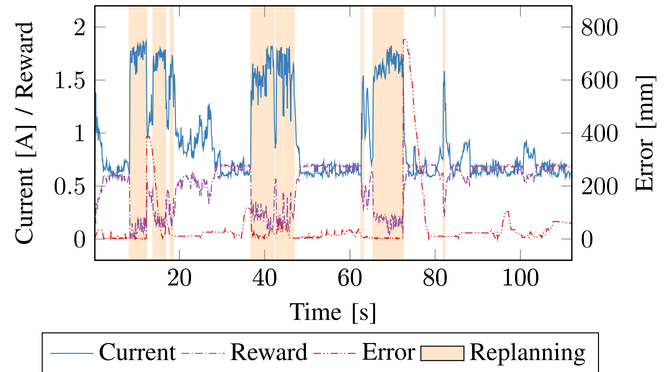

Fig. 4. A time series plot of a single run using the power-only reward function.

The behavior of the power-only reward function in Figure 4 is clear. The learner determines that the best reward comes from aggressively powering down the CPU. Even when error spikes as at ~73 s, the power remains low. This is the correct behavior since the error is due to drift from the ideal path during recalculation. Once the path is calculated, the robot finds the path just about as quickly in low-power as in high-power. Interestingly, this behavior is close to that of the situation-aware governor, however it requires no understanding of the system, nor a tuning of K_p and K_e values.

When we add path error into the reward function as in Figure 5, we see similar behavior as to the power-only. The learner is able to determine that reduction in power brings with it significant reward and therefore attempts to aggressively power-down when not in replanning mode. Even when brief periods of high-error occur, there is little reaction from the

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on June 20,2023 at 23:12:54 UTC from IEEE Xplore. Restrictions apply.

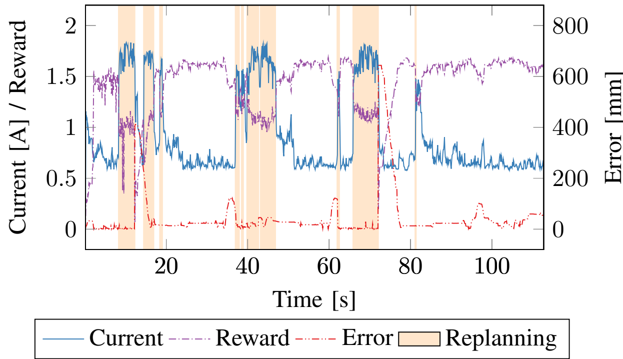

Fig. 5. A time series plot of a single run using a sum of power and error as the reward function.

power manager. Again, because this error is due to path drift during calculation, this can be seen as the right behavior. However, because the path error of a robot cannot be changed instantaneously, and decisions made by the power manager affect the path of the robot only indirectly, we attempt to better balance the near instantaneous changes to power and the longer term changes to path error.

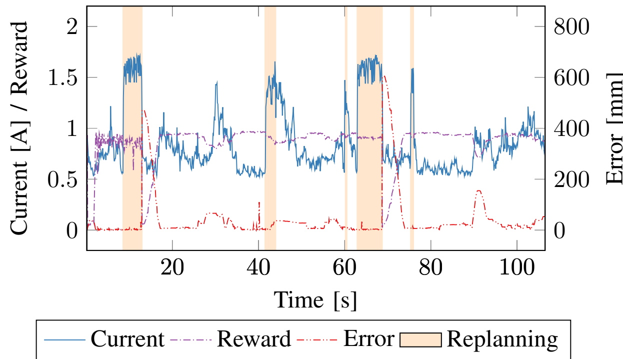

Fig. 6. A single traversal of the course with error and 10-sample moving average of power reward function. $2 - (E_t + \bar{P}_{10})$

In an attempt to better balance the near-instantaneous action-caused changes in power with the slower, less direct changes in error, we introduce an n-sample moving average $\bar{P}_n$ of the power. We examined 10, 50, and 100-sample moving averages in our experiments, however we only present the 10-sample as Figure 6. The 50 and 100-sample MAs performed worse than the 10-sample without any additional noticeable reaction to the error term. Because the sample time is quite short (10 ms), a 10-sample MA still reacts quickly to changes in the CPU states. This allows for the learner to continue to find greater rewards by entering lower power states. However, by losing the near-instantaneous feedback from changing power states, the learning seems to be less effective and thus the reduction of power isn't as significant.

V. Conclusions

In this paper we describe three important outcomes. First, we developed a C-based Q-Learning Quality-of-Service Manager (2QoSM) that takes input from both application and platform to learn a power management policy. The 2QoSM reduced power compared to the Linux on-demand governor by 38.7-42.6% and the situation-aware governor by 4.0-10.2%. Optimizing for minimum error obtained an improvement of 4.6-8.9% over these governors. Moreover, the reward functions, state vector, and learning parameters can be easily adapted for different systems or even dynamically adjusted.

The second important outcome demonstrates the value of software abstractions created by the software framework. The Q-learner was added to existing power management infrastructure without the need to tailor policies to application-specific metrics or states. All that is required from the developer is defining the discretization of desired state variables and assignment of available actions. The abstracted nature of actions allow for any controllable processor or application functions to be adjusted (e.g. powering off cores, scheduling algorithms, or core pinning) to meet the needs of reward functions based on any combination of power, performance, thermal, or reliability metrics. This allows for both quick development of more complex or novel machine learning algorithms and easy porting to different platforms. Moreover, system programmers are not restricted to a specific type of learner, or even to a single paradigm of control at all.

Finally, it is important to recognize the challenge of using even fairly simple machine learning algorithms. Though the mechanism of Q-Learning is well-understood, the learned policies are much less so. There is increasingly a trade-off between ease of use and deployment of high-performing machine learning, and human/operator understanding and tuning of control systems and complex policies.

Future work is ongoing in a number of areas. We are further examining hyperparameters for training and operation. In addition, there are many different potential controlable platform states such as offlining cores, scheduling, or application algorithms. Furthermore, the target policies can be adjusted based upon different reward functions such as total battery life or thermal envelope. Additionally we are examining ML techniques such as $Q(\lambda)$ which may be more suited for significantly time-delayed action-reward pairs. Finally, we are exploring different platforms and applications to test the 2QoSM.

Acknowledgments

This work was funded, in part, by National Science Foundation grant 1538877.

References

[1] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithm for dynamic speed-setting of a low-power CPU," in *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '95. New York, NY, USA: ACM, 1995, pp. 13–25.

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on June 20,2023 at 23:12:54 UTC from IEEE Xplore. Restrictions apply.

[2] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, “Microarchitectural techniques for power gating of execution units,” in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ser. ISLPED ’04. New York, NY, USA: Association for Computing Machinery, Aug. 2004, pp. 32–37.

[3] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, “Flikker: Saving DRAM refresh-power through critical data partitioning,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 213–224.

[4] K. Li, R. Kumpf, P. Horton, and T. Anderson, “A quantitative analysis of disk drive power management in portable computers,” in *Proceedings of the USENIX Winter 1994 Technical Conference*, ser. WTEC’94. USA: USENIX Association, Jan. 1994, p. 22.

[5] Intel Corporation, “Intel automated relational knowledge base (ark),” 2020, [Online; accessed 3-March-2021].

[6] S. Ghose, A. G. Yaglikçi, R. Gupta, D. Lee, K. Kudrolli, W. X. Liu, H. Hassan, K. K. Chang, N. Chatterjee, A. Agrawal, and et al., “What your DRAM power models are not telling you: Lessons from a detailed experimental study,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 3, Dec. 2018.

[7] M. N. Riaz, “Energy consumption in hand-held mobile communication devices: A comparative study,” in *2018 Int’l Conf. on Computing, Mathematics and Engineering Technologies (iCoMET)*, Mar. 2018, pp. 1–5.

[8] G. Bai, H. Mou, Y. Hou, Y. Lyu, and W. Yang, “Android power management and analyses of power consumption in an android smartphone,” in *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov. 2013, pp. 2347–2353.

[9] B. Martinez, M. Montón, I. Vilajosana, and J. D. Prades, “The power of models: Modeling power consumption for IoT devices,” *IEEE Sensors Journal*, vol. 15, no. 10, pp. 5777–5789, Jun. 2015.

[10] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, “Low-power cmos digital design,” vol. 27, no. 4, pp. 473–484, Apr. 1992.

[11] V. Pallipadi and A. Starikovskiy, “The ondemand governor,” in *Proceedings of the Linux Symposium*, vol. 2, 2006, pp. 215–230.

[12] S. Ahmed and B. H. Ferri, “Prediction-based asynchronous CPU-budget allocation for soft-real-time applications,” *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2343–2355, Sep. 2014.

[13] R. Ge, X. Feng, W. c. Feng, and K. W. Cameron, “CPU MISER: A performance-directed, run-time system for power-aware clusters,” in *2007 Int’l Conference on Parallel Processing (ICPP 2007)*, Sep. 2007.

[14] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, “CoScale: Coordinating CPU and memory system DVFS in server systems,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012, pp. 143–154.

[15] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, “RAPL: Memory power estimation and capping,” in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, Aug. 2010, pp. 189–194.

[16] S. Albers and A. Antoniadis, “Race to idle: New algorithms for speed scaling with a sleep state,” *ACM Trans. Algorithms*, vol. 10, no. 2, pp. 9:1–9:31, Feb. 2014.

[17] M. A. Awan and S. M. Petters, “Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems,” in *23rd Euromicro Conference on Real-Time Systems*, Jul. 2011, pp. 92–101.

[18] M. Giardino and B. Ferri, “Correlating hardware performance events to CPU and DRAM power consumption,” in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, Aug. 2016, pp. 1–2.

[19] B. Li and K. Nahrstedt, “A control-based middleware framework for quality-of-service adaptations,” *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 9, pp. 1632–1650, Sep. 1999.

[20] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic, “ControlWare: A middleware architecture for feedback control of software performance,” in *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS’02)*, ser. ICDCS ’02. Washington, DC, USA: IEEE Computer Society, Jul. 2002, pp. 301–310.

[21] H. Hoffmann, “Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems,” in *2014 26th Euromicro Conference on Real-Time Systems*, Jul. 2014, pp. 223–232.

[22] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, “POET: a portable approach to minimizing energy under soft real-time constraints,” in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2015, pp. 75–86.

[23] C. Imes and H. Hoffmann, “Bard: A unified framework for managing soft timing and power constraints,” in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, Samos, Greece, Jul. 2016, pp. 31–38.

[24] M. Giardino, E. Klawitter, B. Ferri, and A. Ferri, “A power- and performance-aware software framework for control system applications,” *IEEE Trans. on Computers*, vol. 69, no. 10, pp. 1544–1555, Oct. 2020.

[25] A. Das, M. J. Walker, A. Hansson, B. M. Al-Hashimi, and G. V. Merrett, “Hardware-software interaction for run-time power optimization: A case study of embedded linux on multicore smartphones,” in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, Jul. 2015, pp. 165–170.

[26] J. F. Martinez and E. Ipek, “Dynamic multicore resource management: A machine learning approach,” *IEEE Micro*, vol. 29, no. 5, pp. 8–17, Sep. 2009.

[27] R. Ye and Q. Xu, “Learning-based power management for multicore processors via idle period manipulation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 7, pp. 1043–1055, Jul. 2014.

[28] H. Shen, Y. Tan, J. Lu, Q. Wu, and Q. Qiu, “Achieving autonomous power management using reinforcement learning,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 2, pp. 24:1–24:32, Apr. 2013.

[29] Y. Ge and Q. Qiu, “Dynamic thermal management for multimedia applications using machine learning,” in *Proceedings of the 48th Design Automation Conference*, ser. DAC ’11. New York, NY, USA: ACM, Jun. 2011, pp. 95–100.

[30] A. Das, B. M. Al-Hashimi, and G. V. Merrett, “Adaptive and hierarchical runtime manager for energy-aware thermal management of embedded systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 2, pp. 24:1–24:25, Jan. 2016.

[31] U. Gupta, S. K. Mandal, M. Mao, C. Chakrabarti, and U. Y. Ogras, “A deep q-learning approach for dynamic management of heterogeneous processors,” *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 14–17, Jan. 2019.

[32] C. J. C. H. Watkins and P. Dayan, “Q-learning,” vol. 8, no. 3, pp. 279–292, May 1992.

[33] R. S. Sutton, A. G. Barto, and R. J. Williams, “Reinforcement learning is direct adaptive optimal control,” *IEEE Control Systems Magazine*, vol. 12, no. 2, pp. 19–22, Apr. 1992.

[34] Chi-Hyon Oh, T. Nakashima, and H. Ishibuchi, “Initialization of q-values by fuzzy rules for accelerating q-learning,” in *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227)*, vol. 3, May 1998, pp. 2051–2056 vol.3.

[35] Y. Song, Y.-b. Li, C.-h. Li, and G.-f. Zhang, “An efficient initialization approach of q-learning for mobile robots,” *Int’l Journal of Control, Automation and Systems*, vol. 10, no. 1, pp. 166–172, Feb. 2012.

[36] E. Even-Dar and Y. Mansour, “Learning rates for q-learning,” *J. Mach. Learn. Res.*, vol. 5, pp. 1–25, Dec. 2004.

[37] DFRobot, “Cherokey 4WD datasheet,” https://wiki.dfrobot.com/Cherokey_4WD_Mobile_Platform__SKU_ROB0102_, accessed: 2021-03-27.

[38] ——, “ROMEO BLE datasheet,” https://wiki.dfrobot.com/RoMeo_BLE__SKU_DFR0305_, accessed: 2021-03-27.

[39] R. Roy and V. Bommakanti, *ODROID XU4 User Manual*, Hardkernel, Gyeonggi, South Korea.

[40] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras, “DyPO: Dynamic pareto-optimal configuration selection for heterogeneous MPSoCs,” *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 123:1–123:20, Sep. 2017.

[41] A. M. Rahmani, B. Donyanavard, T. Mück, K. Moazzemi, A. Jantsch, O. Mutlu, and N. Dutt, “SPECTR: Formal supervisory control and coordination for many-core systems resource management,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. New York, NY, USA: ACM, Mar. 2018, pp. 169–183.

[42] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, “Anytime dynamic a*: An anytime, replanning algorithm,” in *Proceedings of the 15th International Conference on Automated Planning and Scheduling*. AAAI Press, May 2005, p. 262–271.

[43] B. Gough, *GNU Scientific Library Reference Manual - Third Edition*, 3rd ed. Network Theory Ltd.

Authorized licensed use limited to: CAMBRIDGE UNIV. Downloaded on June 20,2023 at 23:12:54 UTC from IEEE Xplore. Restrictions apply.