OptimML: Joint Control of Inference Latency and Server Power Consumption for ML Performance Optimization

GUOYU CHEN and XIAORUI WANG, The Ohio State University, USA

Power capping is an important technique for high-density servers to safely oversubscribe the power infrastructure in a data center. However, power capping is commonly accomplished by dynamically lowering the server processors' frequency levels, which can result in degraded application performance. For servers that run important machine learning (ML) applications with Service-Level Objective (SLO) requirements, inference performance such as recognition accuracy must be optimized within a certain latency constraint, which demands high server performance. In order to achieve the best inference accuracy under the desired latency and server power constraints, this paper proposes OptimML, a multi-input-multi-output (MIMO) control framework that jointly controls both inference latency and server power consumption, by flexibly adjusting the machine learning model size (and so its required computing resources) when server frequency needs to be lowered for power capping. Our results on a hardware testbed with widely adopted ML framework (including PyTorch, TensorFlow, and MXNet) show that OptimML achieves higher inference accuracy compared with several well-designed baselines, while respecting both latency and power constraints. Furthermore, an adaptive control scheme with online model switching and estimation is designed to achieve analytic assurance of control accuracy and system stability, even in the face of significant workload/hardware variations.

CCS Concepts: • **Hardware → Enterprise level and data centers power issues**; • **Computing methodologies → Computational control theory**; **Machine learning**.

Additional Key Words and Phrases: Power capping, latency control, machine learning.

1 INTRODUCTION

Server power capping (i.e., capping the server power consumption within an allowed budget) has been deployed in many data centers to enable safe oversubscription of the power facility, such that a data center can host more servers without expensive upgrade of the power facility (which can range in hundreds of millions USD [14]). Power capping is also an effective way to avoid system failures caused by power capacity overload or overheating due to increasingly high server density. Furthermore, data centers are now facing a well-known *power struggle* problem [40], which can significantly limit their computing capacity. As transistor density continues to increase, a microprocessor chip is expected to soon have more transistors than can be sustainably powered up due to thermal constraints, which is commonly known as dark silicon or the utilization wall [2, 12]. For example, it has been predicted that over 50% of the computing resources may have to stay inactive in 2024 [12]. Hence, it can be expected that most servers in data centers must run under stringent power constraints in the near future.

To date, power capping is accomplished mainly by dynamically adjusting the frequency (and voltage) levels of major server components such as processor [29, 41, 48, 49] and memory [5, 10]. The key reason is that frequency scaling has low overheads and can change the server power consumption promptly to avoid any overload that

This work was supported, in part, by NSF under grant CNS-2336886.

Authors' address: Guoyu Chen, chen.9605@osu.edu; Xiaorui Wang, wang.3596@osu.edu, The Ohio State University, 2015 Neil Ave, Columbus, Ohio, USA, 43210-1272.

ACM 1556-4703/2024/5-ART
https://doi.org/10.1145/3661825

might cause a circuit breaker to trip in seconds [14]. However, lowering frequency to cap the server power consumption can have significant negative impacts on the performance of the applications running on the servers. While such performance degradation might be tolerable for best-effort workloads, such as scientific computing, it can cause the violation of Service-Level Objective (SLO) for applications that have explicit latency requirements.

In recent years, an important application deployed in many data centers is machine learning (or deep learning), which has made significant progress recently and generated new services for cloud computing, such as ML-as-a-Service (MLaaS). For example, major cloud platforms (e.g., Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP)) are already providing a variety of MLaaS, like translation, chatbots, as well as image, video, and text analytics. In such new cloud services, developers need to do two types of work: 1) Perform offline model training that may take a long time to finish and normally has no latency requirements, though on-demand retraining takes much shorter to be done online, and 2) process online inference requests that must be completed in real time with stringent timeliness requirements (e.g., tens of milliseconds per request). Such requirements are often referred to as the SLOs in terms of processing latency [17]. Violating SLOs can result in undesired customer and financial loss and severely hurt the service provider's reputation.

In order to provide SLO guarantees for machine learning (ML) inference requests on data center servers where power capping needs to be strictly enforced, both the server power consumption and the inference latency must be properly controlled. To that end, there can be several possible solutions. The first and immediate solution is to adopt two separate control loops that cap power and reduce latency, respectively, in an independent manner. For example, existing research has already proposed server power capping with dynamic voltage and frequency scaling (DVFS) [41, 48, 49]. Likewise, some other research has tried to control ML inference latency by reducing the required ML computation [4, 15, 27, 39]. Unfortunately, such a separate solution can cause undesired interference between the two control loops, because DVFS can impact ML latency while reduced computation may result in less energy consumption. Such mutual interference can result in system instability and thus power and/or latency violations. The second solution is to carefully coordinate the two control loops by running one loop on a much longer time scale than the other, such that the mutual interference can be minimized and treated as system noise [50, 53]. Unfortunately, neither power nor latency can allow to be monitored and controlled on a long time scale, because violating either constraint can be disastrous. Hence, it is nontrivial to jointly control both server power consumption and ML latency, because it requires a cross-layer solution that can promptly handle both of them.

In this paper, we propose OptimML, a dynamic power and latency control framework that optimizes the online inference performance of ML applications in a data center. To monitor and control both server power consumption and ML inference latency in a timely manner without mutual interference, OptimML features well-established multi-input-multi-output (MIMO) control theory as a design foundation. In sharp contrast to commonly used ad hoc design methodologies, the advantage of such a control-theoretic design is the guaranteed system stability and control accuracy. To our best knowledge, OptimML is the first work that applies advanced MIMO control theory to jointly manage both server power capping and latency control for optimized inference accuracy. Specifically, this paper makes several key contributions:

- We adopt a systematic methodology to model server power consumption and ML inference latency using system identification and validate the models with experimental data.
- We design OptimML based on MIMO control theory for guaranteed system stability and control accuracy. The design choices of each control component are discussed in detail.
- We extend OptimML with an adaptive control scheme that relies on online model switching and estimation to ensure the desired control performance, such as small overshoots/oscillation, despite time-varying ML workloads or different server/GPU hardware.

- We perform extensive experiments on a hardware testbed with three main stream ML frameworks, PyTorch, TensorFlow, and MXNet, to show that OptimML not only achieves the desired control performance, but also outperforms several state-of-the-art solutions by having the best inference accuracy.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 introduces the background on ML latency adaptation. Section 4 provides an overview of OptimML. Section 5 presents the design details. Section 7 analyzes the experimental results. Finally, section 8 concludes the paper.

2 RELATED WORK

There are three categories of ML acceleration work, including structure level, algorithm level, and implementation level [57]. We mainly discuss the structure and algorithm-level solutions because they are most related to our work.

At the structure level, there are different ways to accelerate ML inference with minimized accuracy loss, such as layer decomposition [9, 15, 28], weight pruning [27, 47, 51, 54, 55], channel pruning [39], and block-circulant projection [6]. Most of those solutions are trying to reduce redundant weights that are over-parameterized in a CNN, because a lot of weights are correlated and thus unnecessary. For example, layer decomposition tries to reduce the weight matrix by replacing it with the product of lower-rank matrices in both convolution and fully connected layers of a CNN. Similarly, weight and channel pruning tries to prune unnecessary weights and lowly active channels. While those solutions can effectively accelerate ML inference, they need to change the CNN model structure in an offline manner and thus *cannot* be directly used to dynamically trade off the latency and accuracy of ML inference at runtime.

At the algorithm level, different algorithms have been proposed to speed up the inference computation, such as Fast Fourier Transform (FFT) [27, 31], Winograd [27], and im2col [47]. For example, in FFT, parts of a neural network are converted to the frequency domain for faster computation [31]. The Winograd algorithm can also accelerate convolution by reducing multiplications and increasing additions [27]. Additionally, the im2col algorithm can linearize a matrix into vectors for more efficient computation [47]. Those optimization algorithms are not designed for online adaptation and are orthogonal to our work.

Our work differs from all the aforementioned studies mainly because we try to control the system power consumption, along with inference latency. Although some work optimizes the system energy for ML applications mainly by shortening their execution times [52, 54, 55], they are not designed to cap the power consumption within a budget for safe oversubscription of the power facility.

Power capping has been previously studied for both data centers [29, 40, 48, 49] and chip multiprocessors [21, 32]. For example, Lefurgy et al. [29] have proposed a feedback controller that controls the server power consumption by adjusting the CPU frequency. Ma et al. [35] have proposed a holistic solution to achieve energy efficiency on a GPU-CPU heterogeneous architecture. Wang et al. [53] have developed a solution that adopts control theory to meet the SLO requirements of web services and optimize the power consumption of CPU servers. Savasci et al. [43] have proposed a design of Proportional-Integral (PI) power controller to manage power allocation of servers based on the SLO of web service workloads. Lo et al. [34] have proposed a solution to cap power using an Ad-Hoc controller with respect to multiple SLO thresholds of online and data-intensive workloads. However, those solutions are designed mainly for general best-effort workloads running on CPU and they did not consider the online adaptation of workloads for more fine-grained SLO guarantees. Thus, they cannot provide any explicit SLO guarantees for ML inference running on GPU. In contrast to existing work, we design a MIMO controller for joint power capping and latency control specifically for servers running ML inference services, based on advanced control theory. We also explore ML adaptation such that the SLO can be guaranteed at the application level. In addition, OptimML can adapt to significantly different server hardware and ML workloads by updating the system model online.

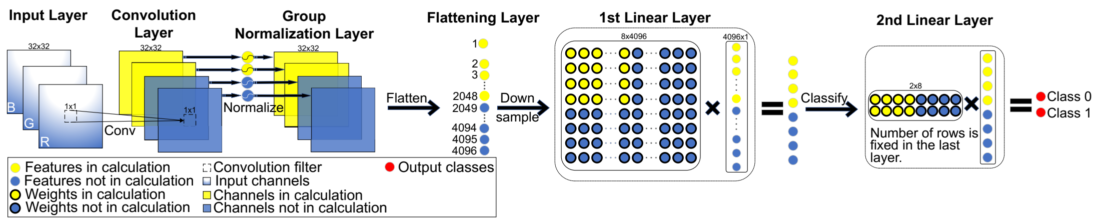


Fig. 1. Model slicing with a slice rate of 0.5. Only the activated parameters and groups (highlighted in yellow or light color) of the existing layer are required in memory and participate in computation.

3 BACKGROUND ON ML ADAPTATION

In this paper, we have investigated three methodologies for online ML inference adaptation: Compression, ensemble, and slicing. First, model compression, such as pruning, layer decomposition, block-circulant projection, mainly tries to reduce the model size and thus the storage space used for weights. However, they are not designed to reduce inference time, because they require iterative fine-tuning and has little support for inference time control [4]. Second, ensemble modeling tries to have many separately trained models, such that different inference latency can be achieved by scheduling different models [23, 24, 44, 51]. However, ensemble modeling significantly increases the requirement of storage. Also, the runtime overhead of scheduling these models can be considerable. The third methodology, model slicing [4], can dynamically change the width of a CNN model (i.e. the number of channels of a convolution layer, and the number of weights in other layers) at runtime, which allows flexible adaptation and control. Specifically, a single parameter called slice rate (between 0 and 1) is used to adapt the model width, and thus the inference time.

Figure 1 shows an example of applying model slicing to a CNN model. We assume that an image with a dimension of $32 \times 32 \times 3$ is the input for classification. The CNN model includes one convolution layer, which expands the three RGB channels of the image into four channels to learn features (with an example filter of 1×1). Then, the group normalization layer normalizes the convolution result to stabilize the mean and variance during inference (each group corresponds to one channel in this example). After normalization, all the features in the four channels are flattened to a vector of 4096 ($32 \times 32 \times 4$) features, which are then downsampled into eight features with an 8×4096 weight matrix in the first linear layer. Finally, the eight features are mapped into two different classes with a 2×8 weight matrix in the second linear layer.

When the slice rate is 1, all the channels, weights, and features are involved in the ML inference computation. If the slice rate is 0.5, in the convolution layer, only the first half of sorted channels (i.e., 2 instead of 4 channels) participate in the computation and normalization. Consequently, only 2048 features are active in the flattening layer. Then, only the first half of the sorted features in the two linear layers are used for ML inference. In Figure 1, those channels, features, and weights that are used in the computation are highlighted in yellow (lighter color). Therefore, with model slicing, the inference time can be dynamically adapted because the amount of computation can be adjusted at runtime.

Model slicing has several advantages for online ML inference adaptation. First, it offers a single knob (i.e., slice rate) to adjust inference time based on a given latency set point, which makes our control design easier. Second, it achieves low-cost inference using a single ML model that includes several subnets, instead of creating a large number of different models. Third, its online adaptation has a small overhead, which does not affect much the controlled latency and power consumption. In our experiment, the switching time between different sizes of subnets is negligible (i.e., less than 2 ms) when the processor runs at full speed. Hence, we choose to use model slicing for online ML inference adaptation in this work, but it is important to note that the proposed control solution can be applied to other adaptation methods as well.

4 SYSTEM ARCHITECTURE DESIGN

In this section, we provide a high-level description of the system architecture of OptimML. The design objective is to control both the average ML inference latency and the system power consumption within their respective limits. The purpose of power capping is to enable safe power oversubscription for reduced capital expenses and prevent power capacity overload or overheating, while the goal of latency control is to provide SLO guarantees for customers. As discussed in Section 1, two separate control loops would be the most intuitive solution, but would result in undesired mutual interference and system instability. A coordinated solution that runs the two loops at different scales would require one control loop to have a slow response. Thus, a multi-input-multi-output (MIMO) control solution must be designed.

There are different system knobs that can be manipulated to control power consumption or/and latency. Those knobs include voltage and frequency of CPU and GPU mentioned in Section 1, and ML model compression, ensemble and slicing mentioned in Section 3. The first one we choose to use is the frequency of the GPU equipped to process the ML computation, even though most prior power capping solutions rely mainly on CPU DVFS. The reason is that ML workloads are processed mostly on the GPU after the data are transferred to GPU memory, while the CPU utilization stays at a low level after the ML workload starts. To verify our hypothesis, we measure the average CPU and GPU utilization for typical ML workloads. Our result shows that the average GPU utilization is around 90%, while the average CPU utilization is only about 30%. Hence, scaling the CPU frequency does not have much impact on the system power consumption. Also, we do not scale the GPU voltage because voltage scaling is not available for most of today's GPUs [22].

The second knob we choose to use is model slice rate, which is discussed in detail and compared with other ML adaptation knobs in Section 3. Slice rate is an application-level knob that can have direct impact on the ML inference latency.

As shown in Figure 2, OptimML features a MIMO control loop that uses two *manipulated variables*, i.e., GPU frequency level and ML model slice rate, to control two *controlled variables*, i.e., system power consumption and ML inference latency. Again, a MIMO controller is necessary because of the coupling among the manipulated and controlled variables. Specifically, a lower GPU frequency can lead to a longer latency, while a smaller slice rate can also affect power consumption. The control loop is invoked periodically and its period is selected such that multiple ML inference requests can be received during a control period and the actuation overhead is acceptable. The following steps are invoked at the end of every control period: 1) The *inference latency monitor* measures the average response time in the last control period and sends the value to the controller. Meanwhile, OptimML also reads the average system power consumption in the last control period from the power meter. 2) Based on the differences between the sampled latency and power consumption and their respective limits (or set points), the controller calculates the slice rate and GPU frequency to be used in the next control period. 3) Then, the slice rate modulator enforces the new slice rate in the ML service, while the frequency modulator sets the new GPU frequency using the toolkit provided by the GPU. 4) The online model estimator updates the system model used by the controller based on the measured power, average response time, and the corresponding GPU frequency and slice rate.

A unique advantage of OptimML is its theoretical foundation based on well-established multi-input-multi-output (MIMO) control theory. While traditional performance/power adaptation solutions heavily rely on heuristic-based design methodologies, the advantages of such a control-theoretic design include 1) guaranteed system stability, 2) better control accuracy, and 3) faster convergence and smaller overshoot. In addition, with control theory, we can also have standard approaches to choosing the right control parameters, such that exhaustive iterations of manual tuning and testing can be avoided. Furthermore, even when workload variations can result in modeling errors, quantitative analysis can be conducted to provide robust control. The controller gain scheduler introduces an extra level of robustness control if the workload ML model varies. As shown in our

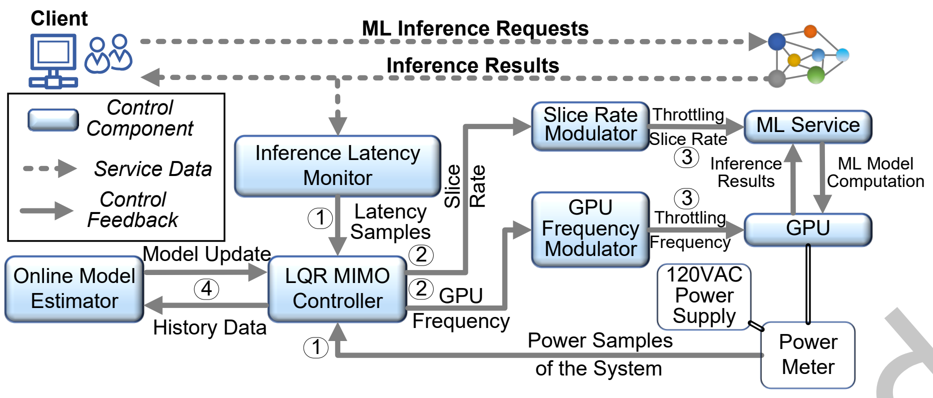


Fig. 2. System architecture of OptimML.

experimental results in Section 7, oftentimes a commonly adopted ad hoc solution can only switch between two discrete GPU frequency levels (e.g., 3 and 4), resulting in a GPU frequency of 3.5 on average. In contrast, OptimML can precisely calculate the ideal GPU frequency (e.g., 3.2) and approximate it with a first-order delta-sigma modulator (i.e., a sequence of 3, 3, 3, 3, 4) [29]. This more accurate control is one of the reasons that OptimML can achieve better inference accuracy.

5 OPTIMML CONTROLLER DESIGN

In this section, we introduce the details of the OptimML controller design. Our control objective is: Whenever the system power consumption or inference latency (i.e., the two controlled variables) deviates from their respective set points, we dynamically adjust both the GPU frequency and slice rate (i.e., the two manipulated variables) in each control period, such that power and latency can converge back to their set points within a finite settling time.

5.1 Problem Formulation

The control of system power and inference latency can be formulated as a dynamic optimization problem. We first define some notation. Let $p(k)$, $l(k)$, $f(k)$, and $s(k)$ be the power consumption, inference latency, GPU frequency, and slice rate of the system in the k^{th} control period. To facilitate the design, let $\bar{p}$, $\bar{l}$,$\bar{f}$, and $\bar{s}$ be their operating points, i.e., their states at a specific time of the dynamic system. We choose $\bar{f}$ and $\bar{s}$ such that $\bar{p}$ and $\bar{l}$ achieve their average values using a typical workload. $\mathbf{u(k)}$ is the vector of the input (i.e., manipulated) variables, i.e., $\mathbf{u(k)} = [\Delta f(k) \quad \Delta s(k)]^T$, where $\Delta f(k) = f(k) - \overline{f}$ and $\Delta s(k) = s(k) - \bar{s}$. Note we use $\Delta f(k)$ and $\Delta s(k)$ (instead of $f(k)$ and $s(k)$ directly) as the control input in our design, such that the system can be linearized. Similarly, $\mathbf{y(k)}$ is the vector of the output (i.e., controlled) variables, i.e., $\mathbf{y(k)} = [\Delta p(k) \quad \Delta l(k)]^T$, where $\Delta p(k) = p(k) - \overline{p}$ and $\Delta l(k) = l(k) - \bar{l}$. $\mathbf{r}$ is the reference point vector of the closed-loop system, which captures the differences between the operation points and the desired power and latency set points P_s and L_s, i.e., $\mathbf{r} = [P_s - \overline{p} \quad L_s - \bar{l}]^T$, and they are constants that can be configured by the data center operator.

Given a reference vector $\mathbf{r}$, the control goal in the k^{th} period is to dynamically choose an input vector $\mathbf{u(k)}$ that minimizes the difference between the output vector $\mathbf{y(k+1)}$ and the reference vector $\mathbf{r}$ in the next control period. We define this difference as the control error: $\mathbf{e(k+1)} = \mathbf{r} - \mathbf{y(k+1)} = [P_s - p(k+1) \quad L_s - l(k+1)]^T$. Specifically, the optimization problem can be formulated as:

$$\min_{\mathbf{u(k)}} \quad (P_s - p(k+1))^2 + (L_s - l(k+1))^2 \tag{1}$$

5.2 System Modeling

In order to have an effective controller design, we must model the dynamics of the controlled system, namely the relationship between the two controlled variables $\mathbf{y(k)}$ (i.e., system power consumption and latency) and the two manipulated variables $\mathbf{u(k)}$ (i.e., GPU frequency and slice rate). Unfortunately, a well-established physical equation is usually unavailable for computer systems. Therefore, we use a standard approach to this problem called system identification [13]. Specifically, we infer this relationship by collecting experimental data and establishing a statistical model.

According to control theory [20], we use the following difference equation to model the controlled system:

$$\mathbf{y(k)} = \sum_{\mathbf{i=1}}^{\mathbf{m_1}} \mathbf{A_i y(k-i)} + \sum_{\mathbf{i=1}}^{\mathbf{m_2}} \mathbf{B_i u(k-i)}, \tag{2}$$

where m_1 and m_2 are the orders of the input and output, respectively. $\mathbf{A_i}$ and $\mathbf{B_i}$ are the control parameters to be determined in the system identification.

In system identification, we need to first determine the orders of the system, i.e., m_1 and m_2. Normally, for the order values, there exists a trade-off between model simplicity and modeling accuracy. Hence, we test different values of m_1 and m_2 (both from 1 to 3) to find the best combination. For each combination of order values, we use sinusoidal sequences of the control input (i.e., $\mathbf{u(k)}$) to stimulate the system and then measure the control output (i.e., $\mathbf{y(k)}$) in each control period. Our experiments are conducted on the testbed introduced in detail in Section 7. Based on the collected data, we apply *Least Squares Method (LSM)* to estimate parameters $\mathbf{A_i}$ and $\mathbf{B_i}$. Then, we estimate the accuracy of the model in the tested order combination, in terms of Root Mean Squared Error (RMSE), and the results are shown in Table 1. Our results show that the order combination of $m_1 = 1$ and $m_2 = 1$ has a small modeling error while keeping the orders (and thus complexity) low. We then use a step-like signal to validate the results of system identification. Figure 3 shows that the estimated output of the selected model is sufficiently close to that of the actual system. Hence, the system model used in our control design is:

$$\mathbf{y(k)} = \mathbf{A_1 y(k-1)} + \mathbf{B_1 u(k-1)}, \tag{3}$$

where $\mathbf{A}$ and $\mathbf{B}$ are 2×2 constant control parameter matrices determined by system identification.

Table 1. System identification results (RMSE) under different order combinations.

	$m_1 = 1$	$m_1 = 2$	$m_1 = 3$
$m_2 = 1$	0.95774	0.95779	0.95785
$m_2 = 2$	0.95779	0.95789	0.95794
$m_2 = 3$	0.95786	0.95802	0.95804

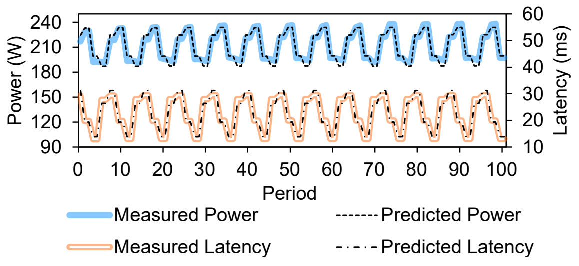


Fig. 3. Measured and predicted power and latency.

5.3 LQR Control Design

We choose to apply the Linear Quadratic Regulator (LQR) control theory [11] to design our controller based on the system model (3). LQR is an advanced control technique that can deal with coupled MIMO control problems. Compared with other MIMO control techniques, such as Model Predictive Control (MPC) that needs to solve a complex optimization problem online in every control period, LQR has smaller runtime computational overheads, which is only a quadratic function of the number of controller variables. We first derive a state-space model based on our system model (3). We use the output vector $\mathbf{y}(\mathbf{k})$ as the nominal state.

In order to track the reference points in vector $\mathbf{r}$, we should not only use the output state $\mathbf{y}(\mathbf{k})$, but also the control error $\mathbf{e}(\mathbf{k})$ [56]. Hence, we consider accumulated control errors to improve the control performance, which is:

$$\mathbf{x_I}(\mathbf{k}) = \mathbf{x_I}(\mathbf{k}-\mathbf{1}) + \mathbf{e}(\mathbf{k}-\mathbf{1}), \tag{4}$$

where $\mathbf{x_I}$ is the integrated error. The final state-space model is

$$\begin{bmatrix} \mathbf{y}(\mathbf{k}) \\ \mathbf{x_I}(\mathbf{k}) \end{bmatrix} = \begin{bmatrix} \mathbf{A_1} & \mathbf{0} \\ -\mathbf{I} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{y}(\mathbf{k}-\mathbf{1}) \\ \mathbf{x_I}(\mathbf{k}-\mathbf{1}) \end{bmatrix} + \begin{bmatrix} \mathbf{B_1} \\ \mathbf{0} \end{bmatrix} \mathbf{u}(\mathbf{k}-\mathbf{1}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \mathbf{r}, \tag{5}$$

Then, we design the LQR controller by minimizing the following quadratic function with respect to $\mathbf{u}(\mathbf{k})$ [20] to compute the controller gain $\mathbf{F}$:

$$\mathbf{J} = \sum_{\mathbf{k}=\mathbf{0}}^{\infty} \begin{bmatrix} \mathbf{y}(\mathbf{k})^{\mathbf{T}} & \mathbf{x_I}(\mathbf{k})^{\mathbf{T}} \end{bmatrix} \mathbf{Q} \begin{bmatrix} \mathbf{y}(\mathbf{k}) \\ \mathbf{x_I}(\mathbf{k}) \end{bmatrix} + \mathbf{u}^{\mathbf{T}}(\mathbf{k})\mathbf{R}\mathbf{u}(\mathbf{k}), \tag{6}$$

where $\mathbf{Q}$ and $\mathbf{R}$ are weighting matrices that determine the trade-off between the control error and the control gain. Specifically, $\mathbf{Q}$ quantifies the cost of state variables diverging from the operating point, and a larger $\mathbf{Q}$ leads to faster response to workload variations. $\mathbf{R}$ specifies the cost of control effort, and the closed-loop system is less sensitive to system noise with a larger $\mathbf{R}$. On our testbed, we set both Q and R to be 1000. Additionally, the first term in (6) represents the control errors and accumulated errors. After minimizing the first term, the system converges to the set point. The second term indicates the control penalty. Minimizing the second item ensures that the controller will minimize the changes in the control input, i.e., the GPU frequency and slice rate of the CNN model, which helps the system stability.

There are different tools to solve the optimization problem (6). For example, a MATLAB command `lqi` can be called to derive the controller gain as:

$$\mathbf{F} = [\mathbf{K_{PD}} \quad \mathbf{K_I}], \tag{7}$$

where $\mathbf{K_{PD}}$ and $\mathbf{K_I}$ represent *proportional and derivative* and *integral* feedback gains, respectively. The dimension of each matrix is 2×2. Hence, given the control errors, we can calculate the input in each control period as:

$$\mathbf{u}(\mathbf{k}) = -\mathbf{F}[\mathbf{y}(\mathbf{k}) \quad \mathbf{x_I}(\mathbf{k})]^{\mathbf{T}}. \tag{8}$$

Based on system identification conducted on the testbed introduced in detail in Section 6.1, our system model (3) has the following parameters:

$$\mathbf{A} = \begin{bmatrix} 0.07473 & -0.01853 \\ 0.00103 & 0.00457 \end{bmatrix}, \ \mathbf{B} = \begin{bmatrix} 1.96101 & -3.97520 \\ -0.32053 & -2.87903 \end{bmatrix}. \tag{9}$$

In our experiments, we use $\mathbf{R} = diag(1000, 1000)$ and $\mathbf{Q} = diag(1000,\ 1000,\ 1000,\ 1000)$. Our resultant LQR controller in (7) has the following parameters:

$$\mathbf{K_{PD}} = \begin{bmatrix} 0.25416 & -0.31881 \\ -0.04142 & -0.16045 \end{bmatrix}, \ \mathbf{K_I} = \begin{bmatrix} -0.22803 & 0.31002 \\ 0.03651 & 0.16045 \end{bmatrix}. \tag{10}$$

The eigenvalues of the closed-loop system are 0.4518, 0.3874, 0.0125, 0.0004, which are within the unit circle. Thus, the system is stable as long as the modeling errors are small. Note that the computation overhead of the designed LQR controller (i.e., (8)) is negligible, because it takes less than one microsecond in each control period on our testbed described in Section 6.1.

5.4 Stability Analysis for Model Variation

One of the key benefits of the control-theoretic design is that the system stability can be analytically guaranteed, even when the working conditions (e.g., workloads) differ from the ones used for modeling. We now discuss how to analyze the stability of the LQR MIMO controller with model variation. We outline the general process below.

1. Derive the control inputs $\mathbf{u}(\mathbf{k})$ based on the *nominal* system model (3).
2. Conduct an automated system identification at runtime to get the actual model parameters $\mathbf{A}_1'$ and $\mathbf{B}_1'$, which are different from $\mathbf{A_1}$ and $\mathbf{B_1}$ in our nominal model (3). The actual model becomes
$$\begin{bmatrix}\mathbf{y(k)}\\ \mathbf{x_I(k)}\end{bmatrix} = \begin{bmatrix}\mathbf{A}_1' & \mathbf{0}\\ -\mathbf{I} & \mathbf{I}\end{bmatrix}\begin{bmatrix}\mathbf{y(k-1)}\\ \mathbf{x_I(k-1)}\end{bmatrix} + \begin{bmatrix}\mathbf{B}_1'\\ \mathbf{0}\end{bmatrix}\mathbf{u(k-1)} + \begin{bmatrix}\mathbf{0}\\ \mathbf{I}\end{bmatrix}\mathbf{r}. \quad (11)$$
3. Derive the closed-loop system model by substituting the derived control inputs $\mathbf{u}(\mathbf{k})$ into the actual system model (11) with $\mathbf{A}_1'$ and $\mathbf{B}_1'$. The closed-loop system model is in the form:
$$\begin{bmatrix}\mathbf{y(k)}\\ \mathbf{x_I(k)}\end{bmatrix} = \begin{bmatrix}\mathbf{A'-K_{PD}} & \mathbf{-B'K_I}\\ -\mathbf{I} & \mathbf{I}\end{bmatrix}\begin{bmatrix}\mathbf{y(k-1)}\\ \mathbf{x_I(k-1)}\end{bmatrix} + \begin{bmatrix}\mathbf{0}\\ \mathbf{I}\end{bmatrix}\mathbf{r}. \quad (12)$$
4. Derive the stability condition of the closed-loop system described by (12). According to control theory, the closed-loop system is stable if all the eigenvalues of the matrix $\begin{bmatrix}\mathbf{A'-K_{PD}} & \mathbf{-B'K_I}\\ -\mathbf{I} & \mathbf{I}\end{bmatrix}$ locate inside the unite circle of the complex space.

In our experiments, we have performed the above analysis to different workloads (e.g., different batch sizes) and testbed. Our results show that the OptimML controller is stable for a wide range of working conditions. In extreme cases where the system might become unstable, model parameters $\mathbf{A_1}$ and $\mathbf{B_1}$ can be dynamically adjusted using the online estimator described in Section 5.5 to match $\mathbf{A}_1'$ and $\mathbf{B}_1'$ for regaining stability.

The following is an example which has different settings from the nominal model. The example workload has a batch size of 100 images instead of 50 images of the nominal model. As a result, the example model has the following parameter:

$$\mathbf{A'} = \begin{bmatrix}0.06122 & -0.02041\\ 0.00126 & 0.00217\end{bmatrix},\ \mathbf{B'} = \begin{bmatrix}1.98531 & -3.93931\\ -0.63155 & -6.04459\end{bmatrix}.$$

We can then calculate the eigenvalues of the matrix described in step 4. The matrices $\mathbf{K_{PD}}$ and $\mathbf{K_I}$ are the same as that of the nominal system. The resulting eigenvalues are 0.4187, -0.1958, $0.0008+0.0044i$ and $0.0008-0.0044i$. Thus, the example system is stable under the control of our controller because all eigenvalues are within the unit circle.

5.5 Adaptive Controller Design

As discussed before, the LQR controller is designed based on an estimated nominal model that may be different from the actual system model, because the controller can be used 1) on servers with different hardware configurations, or/and 2) for ML workloads that may change significantly at runtime. As proven in Section 5.4, the OptimML controller can remain stable when the variations are within certain ranges. However, the LQR controller is designed to achieve optimal control performance, such as small overshoots/oscillation, when the model is accurate. With significant model variations, though still being stable, the LQR controller may have degraded control performance, e.g., large overshoots/oscillations of the controlled variables (*i.e.*, power and latency). In practice, such degradation is commonly undesirable. For example, if a power spike is too much higher than the power cap, it might cause server shutdowns or outages. Therefore, if possible, it is important to dynamically adjust the controller parameters for better control performance if the system model varies at runtime.

In this section, we describe the designs of two adaptive control schemes, both of which are evaluated later to examine their effectiveness. First, we can have several different sets of controller parameters that are tuned for different workloads. This can be practical in some data centers that host only several fixed ML applications that are known as a priori. In such cases, we can have one model for each ML application and switch between different controller models based on our knowledge of the current workload. For example, if a data center operator observes that the current workload has changed from face recognition to speech-to-text conversion, the operator can change the controller to use the parameters designed for the right ML application. We call this scheme *online model switching*.

Second, in some cases, an operator may not have much knowledge of the runtime workload variations. To that end, we can design an *online model estimator* that can automatically and periodically learn from the measured data to update the system model used in the controller. For the convenience of notation, we rewrite the system model (3) in the following form:

$$\mathbf{y}(\mathbf{k+1}) = \mathbf{X}\phi(\mathbf{k}), \tag{13}$$

where

$$\mathbf{X} = [\mathbf{B_1} \quad \mathbf{A_1}],\ \phi(\mathbf{k}) = [\mathbf{u}^{\mathrm{T}}(\mathbf{k}) \quad \mathbf{y}^{\mathrm{T}}(\mathbf{k})]^{T}.$$

We use a Recursive Least Square (RLS) estimator with directional forgetting [33] to estimate and update the parameter matrix **X** of equation (13). The directional forgetting removes the influence of the history data that can affect the system model during parameter updating. Theoretically, directional forgetting searches for a subspace on which the probability distribution remains unchanged by the recently observed data. It reduces the computation overhead of updating the system model and it is proven to be effective and numerically safe [26]. This forgetting factor should be set up by a data center operator. If the workload changes fast, the forgetting factor can be set smaller to reduce the effect of history data. The estimator is defined by the following equations:

$$\begin{aligned}
\hat{\mathbf{X}}(\mathbf{k+1}) &= \hat{\mathbf{X}} + \frac{\epsilon(\mathbf{k+1})\phi^{\mathrm{T}}(\mathbf{k})\mathbf{P}(\mathbf{k-1})}{\lambda + \phi^{\mathrm{T}}(\mathbf{k})\mathbf{P}(\mathbf{k-1})\phi(\mathbf{k})},\\
\epsilon(\mathbf{k+1}) &= \mathbf{y}(\mathbf{k+1}) - \hat{\mathbf{X}}(\mathbf{k})\phi(\mathbf{k}),\\
\mathbf{P}^{-1}(\mathbf{k}) &= \mathbf{P}^{-1}(\mathbf{k-1}) + \left(\mathbf{1} + (\lambda - \mathbf{1})\frac{\phi^{\mathrm{T}}(\mathbf{k})\mathbf{P}(\mathbf{k-1})\phi(\mathbf{k})}{(\phi^{\mathrm{T}}(\mathbf{k})\phi(\mathbf{k}))^{2}}\right)\phi(\mathbf{k})\phi^{\mathrm{T}}(\mathbf{k}),
\end{aligned} \tag{14}$$

where $\hat{\mathbf{X}}(\mathbf{k})$ is the estimate of the true value of **X**, $\epsilon(k)$ is the estimation error vector, $\mathbf{P}(\mathbf{k})$ is the covariance matrix, and λ is the constant forgetting factor with $0 < \lambda \leq 1$. A smaller λ allows the estimator to forget the history data faster. In our experiments, we use $\lambda = 0.5$. The following iteration is invoked in every period of the estimator: 1) The RLS estimator records $\mathbf{u}(\mathbf{k})$ that includes the GPU frequency and the slice rate, and $\mathbf{y}(\mathbf{k})$ that contains the power consumption and the inference latency. 2) The estimator calculates $\mathbf{X}(\mathbf{k})$. 3) The estimator updates $\mathbf{A_1}$ and $\mathbf{B_1}$ in the state-space model (5) of the LQR controller, and the controller gain **F** is re-calculated using the `lqi` command in the Python environment.

The online estimator and the LQR controller are configured to run on different time scales. The estimator is running on a much longer time scale to avoid affecting the stability of the LQR controller. Otherwise, the controller will get incorrect parameter updating if the estimator runs much faster than the controller and the estimator does not receive enough data to do calculation. In our experiments, we enable the online estimator every 5 periods of the LQR controller. As a result, our testbed results show that the LQR controller can stabilize and enter its steady state between every two consecutive invocations of the estimator.

5.6 Implementation of OptimML Control Loop

We now introduce the implementation details of our OptimML MIMO control loop in Figure 2. We implement the LQR controller, the latency monitor, and the power monitor in three separate threads. The online estimator is in the same thread of the LQR controller. We use `numpy` package in Python to implement the RLS calculation and MATLAB-like `lqi` computation for the controller parameters. Because the power meter used in our prototype system has a sampling period of 1s, the control period of all the controllers is set to 5s, such that one period can include multiple power samples to avoid any outliers. Note that the control period can be set to much shorter in a real implementation for faster control response, given a more fine-grained power meter is available.

The inference latency monitor is implemented with the library provided by Nvidia. Before sending a batch of images to the GPU, we record the timing of this CUDA event. After the inference execution, we synchronize this CUDA event and record the elapsed time on the GPU as our inference latency. We focus mainly on the server latency to eliminate the impact of network delays. Since we keep all the tested images locally on the hard drive, we preload all the images (divided into batches) to the memory, in order to minimize the time used to load images.

There exists a trade-off in the selection of the adjustable range of slice rate. The upper bound is 1 (i.e., the full model), while the lower bound is preferable to be small for a wider range, which allows better adaptation. However, the lower bound cannot be too small, because a CNN heavily relies on the base structure to correctly do classifications. Hence, if the slice rate approaches zero, the accuracy may not be acceptable, as discussed in [4]. Based on their results, 0.25 is the best lower bound, which allows the CNN to still have an acceptable accuracy. In addition, between 1 and 0.25, we choose a step size of 1/8 based on the suggestion in [4], resulting in 7 different slice rates {1, 0.8750, 0.75, 0.625, 0.5, 0.375, 0.25}. After receiving a new slice rate from the controller, the inference latency modulator uses it on the next inference job.

The GPU frequency modulator uses the *nvidia-smi* command and the option of application clock, where the value entry is a pair of memory clock and graphic clock. To adjust the frequency levels of the GPU, we fix the memory clock and adjust the graphic clock from 64% (i.e., 562MHz) to 100% (i.e., 875MHz), with an incremental step of 1.5% (i.e., 25 discrete levels). To minimize the impacts of other system components on power consumption, we set all the fans to run at their full speed and the CPU in the performance mode, which can be done with the *cpupower frequency-set -g performance* command.

5.7 Discussion

OptimML is mainly designed for ML-as-a-Service (MLaaS) data centers, where GPUs and servers are rented to customers for training and inference processing of their ML application. Such cloud-based applications usually have ML models that can be handled by a few high-end GPUs. However, OptimML can also be applied to applications with extremely large ML models (e.g., the Transformer model [46]) or small models (e.g., edge computing devices). The Transformer models are recently adopted in large language models, such as text and speech translation and chatBots like ChatGPT [38]. Those models are much larger than the models used in MLaaS. For example, GPT-3 [3] has 175 billion parameters. Such large-scale ML applications have even more urgent demands for power capping and SLO guarantees. Hence, OptimML can be applied to those applications with some extensions to find the best approaches for application-level latency/accuracy adaption. In recent years, ML has also been extensively applied to edge computing devices [1, 16, 30, 42]. Compared to MLaaS applications, edge devices commonly have only limited computing and storage resources [36]. OptimML can be applied to edge devices too, because such devices are commonly powered by batteries and thus also need to limit their power consumption. For example, TensorFlow Lite [45] is a library to deploy ML models on edge devices. OptimML can be used to manage power and latency on edge devices that support TensorFlow Lite.

The length of the control period should be configured by the data center operator based on data sampling rate. The power meter mentioned in Section 6.1 has a sampling rate of 1Hz, and we set the period to be 5s to have enough samples to calculate the average power. The power meter has an accuracy of 1.5%, and our testbed experiments in Section 6 show that the 99th percentile power reading is much less than 2% error range of the power set point. Thus, it is rare that the power meter leads to some outliers. The data center operator needs evaluate the power meter and discard the outlier appropriately especially when the sampling rate is higher.

Tail latency can be used to enforce more stringent latency standards. OptimML uses the average latency in each 5s control period as the controller input and this may result in SLO violations of some batches of the inference workload. Ideally, OptimML can be configured with a much shorter control period (i.e., less than the SLO) to ensure that each batch of the workload meets the SLO if the sampling rate of the power meter is higher. If a data center does not have a power meter with a high sampling rate, the operator can add a safety margin to the set point to ensure that the tail-latency of the ML inference meets the SLO.

When the power-cap value is increased to the amount of an entire center, OptimML can be scaled up at least by the following three ways: 1) One MIMO controller for one server; 2) One MIMO controller for one server rack; 3) One MIMO controller for the entire data center. For the third solution, the computation and communication overhead of one MIMO controller is high, because a data center includes thousands of servers, and this can cause significantly delayed response for both power and latency, resulting in large chances of server shutdown and SLO violations. This overhead can be smaller if the data center adopts the second solution, which has one controller for up to dozens of servers. This design can be used when ML inference tasks need to run on multiple servers in a rack. However, it still has considerable overheads when there are many ML inference tasks running on each server in the rack. When ML inference tasks only run on single servers (i.e., the target scenario in this paper), the most scalable solution would be the first one, where every server has one MIMO controller to have joint control of power and latency. It guarantees the power of that server is within the cap and a smaller chance of SLO violation.

6 TESTBED SETUP AND BASELINES

In this section, we introduce the hardware testbed and baselines used in our experiments.

6.1 Hardware Testbed

Our experiments are conducted on a system with a Nvidia Tesla®K80 GPU, which has 25 overlocked core frequencies if the memory is configured to work at 2505MHz. We measure the system power consumption with a Wattsup PRO power meter, whose power readings are sent to the controller every second through a serial communication. The OS is Ubuntu 20.04 LTS. We follow the experimental setup in [4] to implement model slicing using PyTorch 1.8.2 with CUDA 11.2, where we add ML adaptation feature to convolutional layers, normalization layers and linear layers, which is also mentioned in Section 3. We also evaluate our control framework on TensorFlow 2.4.0 or MXNet 1.9.1 with similar changes. The neural network used for inference is Resnet50 and Resnet28 [19], which are widely adopted in image classification jobs and are trained using the CIFAR-10[7] dataset with 150 epochs and a batch size of 50. The initial learning rate is 0.1, as in [4]. We decrease the learning rate by 10% at the 75th epoch and at the 113th epoch. The training process ensures the appropriate weights and connections under different slice rates. This process is also accelerated with a Tesla V100 GPU provided by Google Colab Pro[8]. We used the source code in [37] to do model slicing.

Metrics. There are several metrics used in our experiments for the comparison between OptimML and the baselines. The first one is control accuracy, which means how closely a solution can control power and latency to their respective set points. Control accuracy is important because violating either the power or latency/SLO constraint can result in undesired system shutdown or bad customer experiences, respectively, as discussed before. However, if a solution simply keeps power/latency way below their set points for safety, it is also bad because doing so may lead to inferior ML inference accuracy, which is our second metric. The reason is that such a headroom can be exploited to run the system at a higher frequency or with a larger slice rate.

In online image classification, the ML inference accuracy is defined as the percent of images that are correctly classified in a batch. A small drop in inference accuracy can make a big difference in many ML applications, such as street object recognition used for autonomous vehicles. Many recent studies have considered their accuracy improvement of 6% or less a significant contribution [6, 18, 25, 39, 51, 55]. In our experiments, we also find that raw accuracy value of an open-loop system (without any control) using Resnet50 is 87.09% when the slice rate is 1. The raw value decreases slightly to 86.95%, 86.52%, 85.92%, 84.16%, 80.98%, and 75.34%, when the slice rate is reduced to 0.8750, 0.75, 0.625, 0.5, 0.375, 0.25, respectively. That means the maximum possible range is only 11.75%. Hence, we use a metric called the relative accuracy, which is defined as the measured raw accuracy minus 75.34% (i.e., when the slice rate is the smallest) and then divided by the range of 11.75%. For the experiments using Resnet28, the raw accuracy minuses 86.23% and the result is divided by 4.75%. In the following, we use relative accuracy in the comparison with the baselines.

6.2 Baseline Designs

We use the baselines below for comparison.

Ad-Hoc: There does not currently exist a solution that controls both power and latency for ML inference. The state-of-the-art solutions either cap power by adjusting frequency (e.g., [41]) or control inference time with slice rate (e.g., [4]). Since both of them are based on heuristic for adaptation, we implement them to show the advantage of control-theoretic designs and call them *Ad-Hoc-Power* and *Ad-Hoc-Latency*, respectively. For each of the two Ad-Hoc solutions, in every control period, it compares the current power/latency value with the desired set point. If the value is higher than the set point, Ad-Hoc simply lowers the frequency/slice rate by one level until the new power/latency value becomes lower than the set point, and vice versa. For Ad-Hoc-Power, because it does not explicitly control latency, it may violate the SLO constraint. One way is to use the smallest slice rate for the shortest possible latency that ensures safety. However, doing so would result in poor ML inference accuracy. Hence, we set the slice rate of Ad-Hoc-Power to be the middle value (i.e., 0.625) for a compromise, but such a setting still could lead to latency violation. Likewise, for Ad-Hoc-Latency, we set its frequency to be the middle value (i.e., 82%).

SISO: Similar to Ad-Hoc, we design two separate Single-Input-Single-Output (SISO) controllers based on the control theory, as *SISO-Power* and *SISO-Latency*. Compared to Ad-Hoc, SISO solutions can achieve more accurate power/latency control, as shown later in our experiments. However, because the two SISO controllers are designed to control either power or latency, they have to set slice rate/frequency to the middle levels (i.e., 0.625 and 82%), as explained for Ad-Hoc. Such middle levels could either result in violations or inferior ML inference accuracy, as can be seen later in our experiments.

We now briefly introduce how the two SISO controllers are designed. SISO-Power adapts the GPU frequency to control the server power consumption, while SISO-latency adapts the model slice rate to control the inference latency. We first need to model the dynamics between the input and the output of the system to design the SISO controller. We can infer the relationship by system identification, which is collecting output data for every GPU frequency and every slice rate, respectively. Then, we build a statistical model according to the measured data. Our results on the testbed show that the models of both controllers are linear, which are shown in Figure 4. Hence, the design is the same for the two SISO controllers and we just present the design process of SISO-Power in the following. The linear system model we have is:

$$p(k+1) = p(k) + A(f(k+1) - f(k)), \tag{15}$$

where $p(k)$ and $f(k)$ are the power consumption and GPU frequency of the system in the k^{th} control period. A is the slope of the system model. Based on the linear model, we can design a proportional (P) controller, whose Z-transform is $C(z) = \frac{1}{A}$. We do not choose a more complex PID (Proportional-Integral-Derivative) controller because of two reasons: 1) The first-order delta-sigma modulator mentioned in Section 4 already has the integral function to smooth the controlled power and latency. 2) The derivative term may amplify the noise in server power and ML inference latency. In the time domain, the GPU frequency in the next control period can be calculated as:

$$f(k+1) = f(k) + \frac{1}{A}(P_s - p(k+1)) \tag{16}$$

where P_s is the desired power set point.

When the system model (15) is accurate, this P controller is proven to be stable with a zero steady-state error. However, as discussed before, the real system model can be different due to workload/hardware variations. As shown in Figure 4, a real system model is also linear but with a different slope. Without loss of generality, we model the real system as:

$$p(k+1) = p(k) + g_1 A(f(k+1) - f(k)), \tag{17}$$

where g_1 is the system gain used to model the variation between the real system model (17) and the nominal model (15). The closed-loop transfer functions for the real system is:

$$\frac{P(z)}{P_s z/(z-1)} = \frac{g_1}{z - (1 - g_1)} \tag{18}$$

We now analyze the stability and steady-state errors when the real system model differs from the nominal model. Based on the transfer function in (18), the system is stable if and only if the pole is within the unit circle, namely, $|1 - g_1| < 1$. Hence, the real system is guaranteed to be stable as long as the model variation is within the range: $0 < g_1 < 2$. We can then derive the steady-state error with

$$\lim_{z \to 1}(z-1)P(z) = \lim_{z \to 1}(g_1 \frac{z}{z - (1 - g_1)})P_s = P_s, \tag{19}$$

Hence, the system is guaranteed to converge to the set point if the system is stable, despite any model variation. For the settling time of the P controller, we can use MATLAB to analyze its settling time. Our result shows that power can settle to the set point within 5 periods even in the worst case.

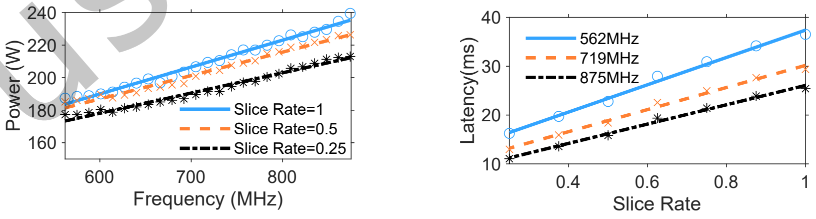


Fig. 4. Both power and latency have a linear model, even when the system settings are different.

P5+P25: We then design a coordinated controller that controls both power and latency. Specifically, it is implemented by running both the two SISO controllers introduced above. However, as discussed before, there exists some mutual interference (i.e., coupling) between the two control loops, because GPU frequency affects latency while slice rate has impacts on power consumption. Hence, P5+P25 is designed to follow the design methodology in [53] to run SISO-Latency in a longer control period, i.e., 25s, because our analysis shows that SISO-Power (whose control period is 5s) has a settling time of 5 control periods. With this coordinated design, SISO-Power can stabilize within one period of SISO-Latency, such that the coordinated controller can achieve the global stability under the nominal model setting. However, this coordinated design still violates the SLO under bursty inference workloads. We will show such a sophisticated design is still inferior to the MIMO control design of OptimML, which explicitly models the mutual interference.

7 EVALUATION RESULTS

We first present the typical results of OptimML and the baselines. We then compare OptimML controller with baselines in terms of control accuracy under different set points. We then test the impacts on ML inference accuracy.

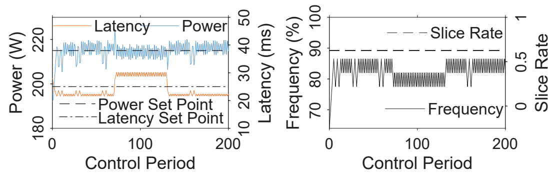
7.1 Typical Runs of OptimML and Baselines

For this set of experiments, we test OptimML and the baselines in a typical scenario for data center servers. We fix the controller gain in these experiments. Initially, system power and latency are to be controlled to their respective set points of 215W and 25ms. In the middle of the run, in the 70th control period, we emulate a workload change by increasing the batch size from 50 images to 70. This can often occur when inference requests are received from a different ML application. In the 130th control period, we change the batch size back to 50 to stress test all the solutions.

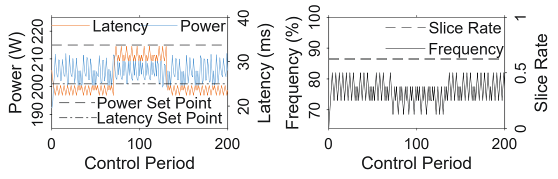
Ad-Hoc: We first evaluate the two Ad-Hoc controllers, which are similar to those solutions used in the industry. Again, Ad-Hoc simply raises or lowers GPU frequency or slice rate by one level, depending on whether the measured power or latency is lower or higher than the set point. Figure 5a shows a typical run of Ad-Hoc-Power. We can see that Ad-Hoc-Power takes multiple control periods to raise power above the set point by increasing the GPU frequency, one level at a time. Afterwards, it lowers the frequency such that power drops below the set point. Then, it oscillates between the two adjacent frequency levels and has an average power value that is higher than the set point, which violates the required power constraint and could trip the circuit breaker and cause undesired shutdown. When the workload changes in the 70th period and causes a high power spike, Ad-Hoc-Power responds by lowering the GPU frequency to get power close to the set point again. Because Ad-Hoc-Power does not control latency, so it uses the middle slice rate (i.e., 0.625) by default, as explained before. As a result, its latency is initially lower than the set point, but increases with the workload and then violates the set point, as shown in Figure 5a. This demonstrates that latency should be controlled together with power.

To avoid power violation, we can add a safety margin to Ad-Hoc-Power, such that even the highest power consumption stays just below the constraint for a particular workload. Figure 5b shows the results of this *Safe Ad-Hoc-Power*, where its highest power value (before the workload change) is just slightly below 215W. However, this safety margin slightly increases the latency, because lower frequencies are used. The latency of Safe Ad-Hoc-Power stays below the set point initially, but violates the set point by a large gap when the workload increases in the middle of the run. Because the original Ad-Hoc-Power causes power violation (even before any workload change) and so cannot be used in practice, in the rest of the paper, we will test Safe Ad-Hoc-Power instead. However, Safe Ad-Hoc-Power also has two problems: 1) it can violate the latency constraint sometimes due to its open-loop nature, and 2) though it can meet the power constraint, its large headroom would result in a poor ML inference accuracy (as shown in a later section), because the GPU frequency is set too low unnecessarily due to the safety margin.

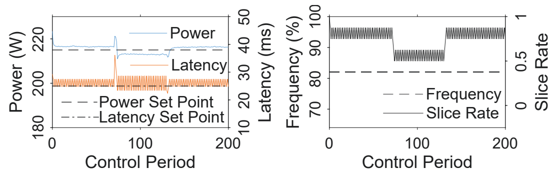
Similarly, Figure 5c shows that Ad-Hoc-Latency also oscillates between two adjacent slice rates because the set point is between their latency values. Consequently, it has an average latency above the set point. Because Ad-Hoc-Latency does not control power, the middle GPU frequency is used by default, which causes power



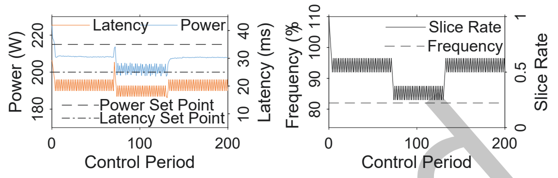
(a) Ad-Hoc-Power.



(b) Safe Ad-Hoc-Power.

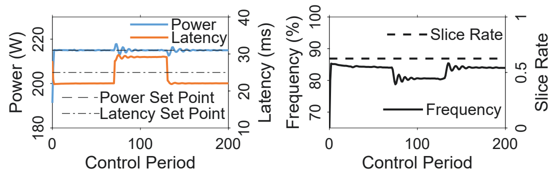


(c) Ad-Hoc-Latency.

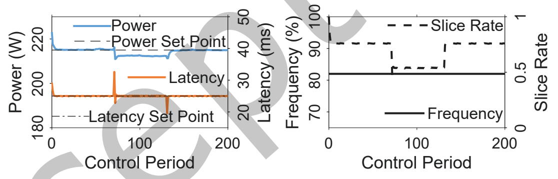


(d) Safe Ad-Hoc-Latency.

Fig. 5. Typical runs of the two Ad-Hoc controllers and their improved variants.



(a) SISO-Power.



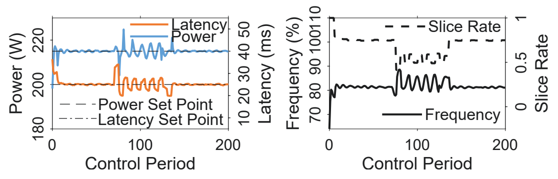
(b) SISO-Latency.

Fig. 6. Typical runs of the two SISO controllers. Compared to Ad-Hoc, SISO has much better control accuracy due to its control-theoretic design.

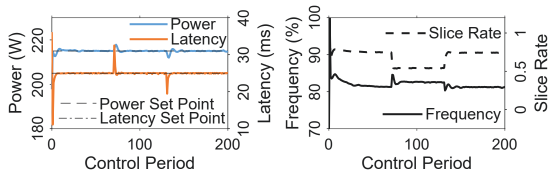
violation, which shows that it is always hard to find a constant GPU frequency that can meet the power constraint. An improved *Safe Ad-Hoc-Latency* with a safety margin (for latency) can ensure both latency and power stay below the set points all the time by using smaller slice rates, as shown in Figure 5d, but might also result in a poor ML inference accuracy.

SISO: We now test the two SISO controllers that are designed based on control theory. Figure 6a shows that the system power under SISO-Power can precisely converge to the set point without much oscillation, which clearly demonstrates the advantage of control-theoretic designs that can produce a fractional frequency level based on its model, instead of having to switch between two adjacent levels. Because SISO-Power does not control latency, we use the middle slice rate, which results in latency violation when the workload increases in the 70th period, just like Ad-Hoc. It is commonly difficult to select a single constant slice rate that always leads to a latency just below the set point, due to the system diversity and time-varying working conditions. This is one of the motivations for feedback-based designs. Figure 6b shows similar results for SISO-Latency.

P5+P25: As introduced before, P5+P25 coordinates the two SISO control loops by running SISO-Latency with a control period of 25s, 5 times that of SISO-Power. As a result, we can see from Figure 7a that both power and latency can be controlled to their respective set points. This is in sharp contrast to Ad-Hoc and SISO that can only control one of the two variables. However, as discussed before, a control period of 25s is too long for a timely response that brings latency to the desired set point both initially and when the workload changes, which is detrimental to ML inference with stringent latency requirements. Hence, it is highly desirable to have a MIMO controller that can control both power and latency in a timely manner without causing much mutual interference.

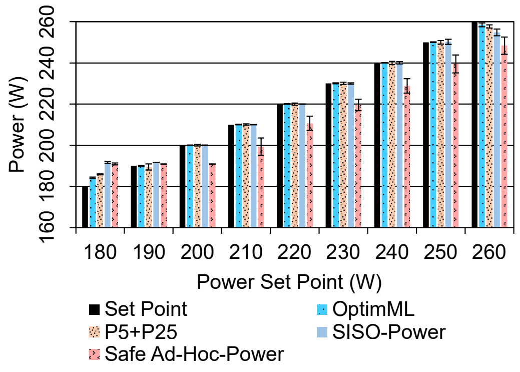


(a) P5+P25.

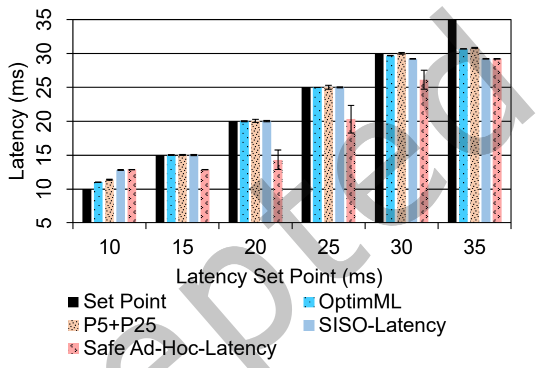


(b) OptimML.

Fig. 7. Typical runs of P5+P25 and OptimML



(a) Different power set points.



(b) Different latency set points.

Fig. 8. Comparison under different power and latency set points. OptimML has the best control accuracy with the smallest standard deviations.

OptimML: A key advantage of OptimML is its MIMO control design based on the LQR control theory. Hence, it is capable of handing the coupling (i.e., mutual interference) between power and latency and control both in the same control period of 5s. Figure 7b shows a typical run of OptimML. We can see that both power and latency can settle within 3 control periods under OptimML, which is faster than all the baselines. In addition, OptimML has the smallest oscillation among all the solutions despite any workload changes and thus a high control accuracy. For example, OptimML temporarily violates the power set point in only two control periods due to the workload increase in the 70th period, while Ad-Hoc-Power constantly violates the power set point in 68 periods, if we consider a 1% tolerance range around the set point. This better control accuracy can be translated to better ML inference accuracy, as to be discussed in Section 7.3.

7.2 Control Accuracy under Different Set Points

In this set of experiments, we test OptimML and the baselines under different power and latency set points to see how accurate their control can be. As discussed before, accurately controlling power/latency can result in better resource utilization and thus better ML inference accuracy. To avoid the impacts of any transient states at the beginning of each run as shown in the last subsection, we calculate the average power/latency and their standard deviation (as error bars) after the system enters the steady state (i.e., the last 90 control periods out of each run of 100 periods in total). We first test different power set points from 180W to 260W with an interval of 10W, while having the latency set point fixed at a typical value of 20ms. Ad-Hoc-Power is excluded from comparison because it can cause power violation, as explained before. Figure 8a shows that Safe Ad-Hoc-Power has the worst control accuracy, because it has a safety margin that ensures its highest power value stays below the set point

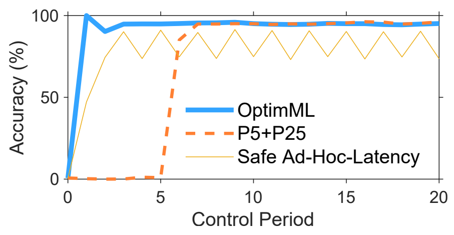


Fig. 9. ML Inference accuracy for OptimML and baselines. OptimML is the fastest to achieve a high accuracy of 95%.

when it just switches between two adjacent GPU frequency levels. Unfortunately, though such a margin ensures safety, it also often leads to unnecessarily lower GPU frequency and thus worse ML inference accuracy. Safe Ad-Hoc-Power also has the largest oscillation and so largest deviation. SISO-Power has better control accuracy than Safe Ad-Hoc-Power due to its control-theoretic design. However, it violates the power constraint when the set point is 190W, because SISO-Power relies only on GPU frequency (without slice rate) to adapt power, so it has a smaller adaptation range than OptimML and P5+P25, which adapt both frequency and slice rate.

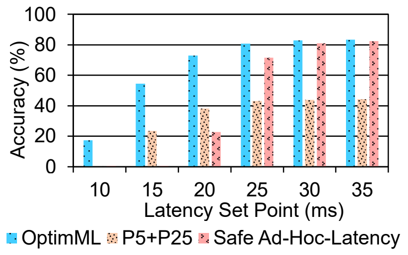
Both OptimML and P5+P25 can accurately converge to almost all the desired set points, except the two border cases (i.e., 180W and 260W), where all the controllers fail. This is because the two set points are out of the adaptation range for all the controllers. For example, OptimML has set the GPU frequency and slice rate to their minimum (i.e., 562MHz and 0.25) when the set point is 180W, but the real power consumption is still around 183W. In practice, it is infeasible to have such a low/high power constraint. OptimML still manages to have the smallest distances from the set point even for those border cases. P5+P25 has the second best control accuracy but its deviations are much greater than those of OptimML.

We then test different latency set points from 10ms to 35ms with an interval of 5ms, while having the power set point fixed at a typical value of 215W. Similar to the results under different power set points, Figure 8b shows that Safe Ad-Hoc-Latency has the worst control accuracy due to its safety margin, which leads to unnecessarily lower slice rates and thus a poor ML inference accuracy. SISO-Latency has a limited adaptation range and so cannot settle to 30ms. OptimML and P5+P25 can accurately converge to almost all the desired set points due to their larger adaptation range, except the two border cases (i.e., 10ms and 35ms) that are out of the range. The control accuracy of P5+P25 is slightly worse than that of OptimML. So, we compare OptimML against P5+P25 in more detail in the next subsection.

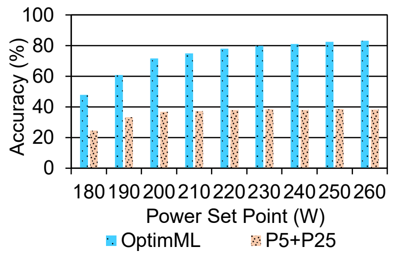
7.3 Detailed Comparison with P5+P25

Because OptimML and P5+P25 perform similarly well, we now examine them in more detail. As introduced before, P5+P25 is designed by coordinating two SISO controller based on control theory, so it has all the advantages of control-theoretic designs. OptimML is designed based on MIMO control theory so it can better handle the coupling (i.e., mutual interference) between power and latency. A key disadvantage of P5+P25 is that it runs the latency SISO control loop 5 times slower than the power loop, in order to achieve global stability, which can result in slower response to latency violation. Now, we show that such a violation could lead to worse ML inference accuracy during a transient state (i.e., before the controllers converge to the set points).

We test a scenario with the power and latency set points at 215W and 25ms, respectively. The initial GPU frequency and slice rate are set to the lowest levels, in order to highlight the differences in the transient state. Figure 9 shows that OptimML quickly converges to the desired set points within two control periods. As a result, it immediately achieves a high ML inference accuracy at 95%. In contrast, because P5+P25 needs to run the latency control loop 5 times slower than the power loop, it has a low accuracy of 0.4302% in the first five periods before taking three more periods to ramp up to 95%. Note again that P5+P25 cannot run the latency loop faster than the power control, because violating power constraint is more dangerous. For comparison, we also test Safe

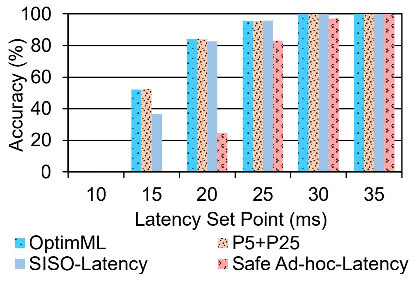


(a) Under different latency set points.

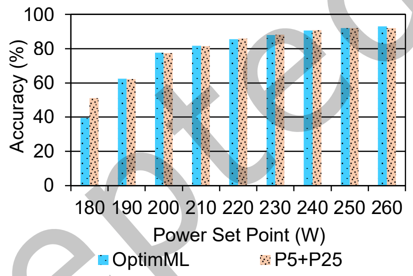


(b) Under different power set points.

Fig. 10. ML Inference accuracy in the transient state. OptimML has the highest accuracy.



(a) Under different latency set points.



(b) Under different power set points.

Fig. 11. ML inference accuracy of OptimML and the baselines in the steady state.

Ad-Hoc-Latency that oscillates between two slice rates and achieves an accuracy of 82% due to its safety margin. One might argue that transient states are short and have small impacts. However, a real system can frequently enter a transient state whenever the working conditions (e.g., workload, set points) change considerably, so the transient-state performance is important too.

We now test OptimML and P5+P25 under different latency and power set points. Figure 10a shows the results when the latency set point increases from 10ms to 35ms with the power set point fixed at 215W. We focus on the transient states by presenting the average accuracy during the first 10 control periods. We can see that OptimML achieves the highest ML inference accuracy in all the cases. Its accuracy increases rapidly from 10ms to 25ms, but stays the same from 30ms to 35ms. That is because all the controllers cannot settle to 35ms, as shown earlier in Figure 8b. Figure 10b shows the results when the power set point increases from 180W to 260W with the latency set point fixed at 20ms. Similarly, OptimML has a much higher accuracy than P5+P25 because of its faster convergence. We do not include Safe Ad-Hoc-Power here because it does not adapt slice rate and so has no effect on ML inference accuracy.

7.4 Comparison of ML Inference Accuracy

After testing the ML inference accuracy in the transient states, we now evaluate OptimML and the baselines in terms of their accuracy in the steady states (i.e., the last 90 periods of each 100-period run). We first fix the power set point at 215W and test different latency set points from 15ms to 35ms. Figure 11a shows that Safe Ad-Hoc-Latency has the worst accuracy due to its safety margin. At 15ms set point, SISO-Latency has a lower accuracy than OptimML and P5+P25 because it relies only on slice rate to reduce latency, while the other two can reduce the GPU frequency as well, which has smaller impacts than slice rate on inference accuracy. As discussed

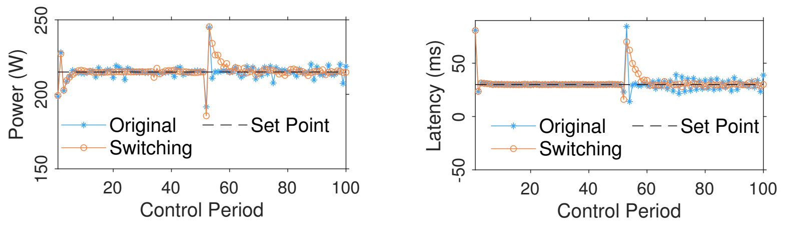


Fig. 12. Online Model Switching has much less oscillation of power and latency than the Original OptimML after the workload change in the 50th control period.

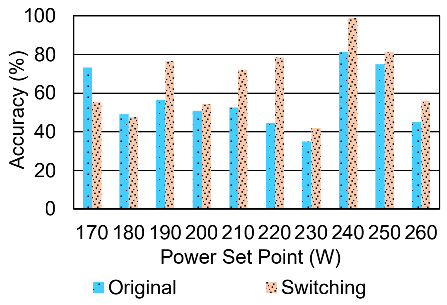
before, P5+P25 performs worse than OptimML in the transient state, resulting in its slightly lower accuracy. Figure 11b shows similar results when the power set point increases from 180W to 260W, with the latency set point fixed at 20ms. We do not include Ad-Hoc-Power and SISO-Power here because they do not adapt slice rate and thus have no impact on accuracy. OptimML has a higher accuracy than P5+P25 in all cases except the border case of 180W, where both controllers cannot settle to 180W as discussed in Section 7.2. Although it seems that P5+P25 has a higher accuracy than OptimML at 180W, this is actually caused by P5+P25 having a worse control accuracy and so violating the 180W set point more than OptimML, as clearly shown in Figure 8b.

7.5 Adaptive Control with Online Model Switching and Estimation

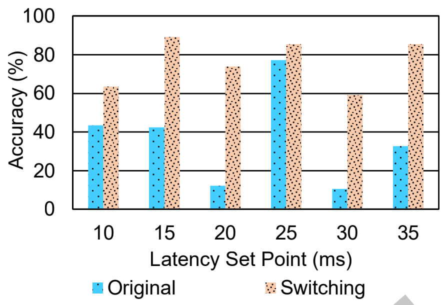
In all the previous experiments, OptimML has used the same control parameters designed based on the nominal system model (3). As discussed in Section 5.5, in a cloud computing data center, a GPU server may host several ML applications that have different system models, such as face recognition and speech-to-text conversion. Therefore, when the operator observes an application change at runtime, online model switching can be adopted to achieve better control performance. For a GPU server with unknown application changes, an online model estimator can be used to automatically adjust the control parameters based on the measured server power consumption and inference latency.

Online Model Switching. We first test model switching by focusing on a typical scenario where the ML workload is changed from Resnet50 to Resnet28 in the middle of the run. The power set point is set to 215W and the latency limit is 30ms. The original OptimML uses the same control parameter **F** derived for Resnet50 throughout the entire 100 control periods. With online model switching, after observing the workload change, we dynamically switch the control parameter to the one designed for the new workload Resnet28 in the 50th control period. Figure 12 shows that online model switching results in much smaller oscillation than the original OptimML. In ML applications, it is always preferable to have smaller latency oscillation. For example, even though the average latency is controlled to meet the SLO, higher latency spikes can indicate some customers have experienced longer response time. For power capping, a high spike can have a higher chance of causing the circuit breaker to trip for outages.

We now examine the impact of adaptive control on ML inference accuracy. Because adaptive control is designed to achieve better control performance when the controlled variable deviates from the set point due to system model variation, the main difference between an adaptive controller and the original controller is the transient state when the controlled variable settles back to the set point. Hence, we compare their inference accuracy during such transient states (from the 50th to the 60th control period) when the system model is changed from Resnet50 to Resnet28 in the 50th control period. We first measure the average accuracy of the original OptimML and the one with online model switching, when the power set point increases from 170W to 260W with a step size of 10W and the latency set point is fixed at 20ms. Figure 13a shows that the inference accuracy is up to 33.85% higher with online model switching. Note that OptimML seems to have higher accuracy when the power



(a) Under different power set points.

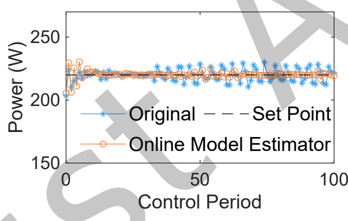


(b) Under different latency set points.

Fig. 13. Inference accuracy comparison between the original OptimML and Online Model Switching during transient states.

set point is 170W and 180W, because the original controller saturates at values higher than the set points and violates both the power and latency limitations. The result in Figure 13b shows that the inference accuracy is up to 52.97% higher with online model switching, when the latency set point increases from 10ms to 35ms with a step size of 5ms. This set of experiments demonstrate that online model switching can help achieve better control performance and inference accuracy if the application models are known ahead of time.

Online Model Estimator. When the ML application model can change unexpectedly at runtime, an online model estimator can be used to dynamically adjust the controller parameters based on the measured power and latency values. In this experiment, we examine the online model estimator by focusing on a typical scenario where we use the original OptimML controller that is designed based on the model of Resnet50 to control a different workload, Resnet28. If we know the real workload is Resnet28, online model switching could have been used like before. But here we assume we do not have such knowledge of the workload and so have to use online model estimator for adaptive control.



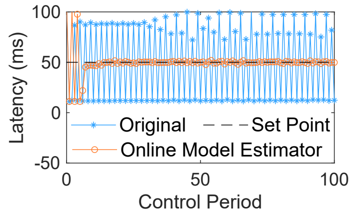


Fig. 14. Online model estimator can help achieve better control performance even when the system model differs significantly from estimation

Figure 14 shows that the original OptimML has large oscillations, though the average power and latency have been successfully controlled to the set points, *i.e.*, 220W and 50ms. Although OptimML has a zero steady-state error, the high spikes are generally undesirable as discussed before. For example, in the 50th period, the power peak is 8.63W higher than the set point, which could trip the circuit breaker if the set point is set to be exactly the breaker capacity. The standard deviation of the measured power values is 4.93W after the 10th period. For latency, high spikes indicate long response time for some customers. The standard deviation of latency is 38.56ms. We then enable the online model estimator and rerun the experiments with the same initial parameter values in the OptimML system model. Figure 14 shows that the estimator can quickly correct the system model at runtime in just several control periods from the beginning. As a result, OptimML (with estimator) performs as well as before when the system model is accurate, with a standard deviation of only 1.27W and 1ms. This experiment demonstrates that the online model estimator is important to the OptimML LQR controller, especially 1) when the workload is unknown or could vary significantly at runtime and 2) when the controller is used to control GPUs with different processing capacities.

7.6 Effectiveness of OptimML on TensorFlow and MXNet

All the previous experiments so far are conducted with the widely adopted ML framework, PyTorch. Since there are some other major ML frameworks such as TensorFlow from Google and MXNet from Apache. In this section, we evaluate OptimML on the TensorFlow and MXNet frameworks to show OptimML does not depend on a particular ML framework. We test the original OptimML that is designed based on PyTorch here, because the previous section already demonstrates the advantage of adaptive control.

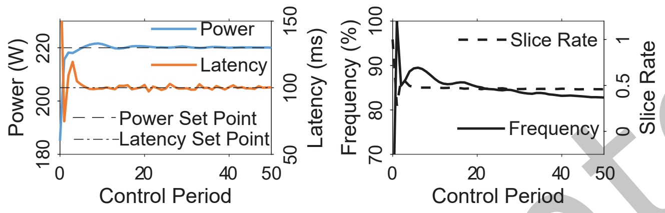
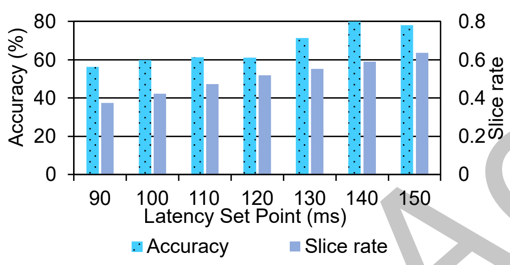
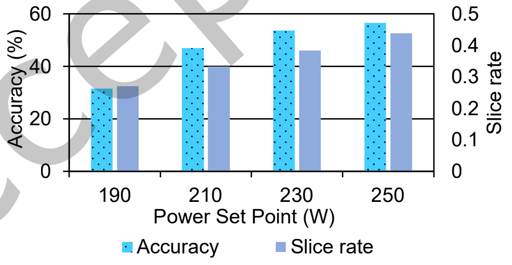


Fig. 15. Typical runs of OptimML on TensorFlow. Both power and latency are successfully controlled to their set points.



(a) Under different latency set points.



(b) Under different power set points.

Fig. 16. ML inference accuracy of OptimML on TensorFlow in the steady state.

TensorFlow. We first test OptimML on Tensorflow, which can be used for various ML/AI applications, particularly for training and inference of deep neural networks (DNNs). In this set of experiments, the ML workload is still Resnet50 with an image batch size of 250. The power set point is set to 220W. The latency set point is at 100ms. We run the experiments for 50 control periods. Figure 15 shows that both power and latency still settle to their respective set points within 5 control periods even if we run OptimML on TensorFlow. This is important because a data center operator can use OptimML to ensure power safety and SLO guarantees without having to rely on a certain ML framework. Then, we evaluate OptimML under different power and latency set points to examine how the inference accuracy varies when running on TensorFlow. We first fix the power set point at 200W and change the latency set point from 90ms to 150ms. To exclude transient stages, the inference accuracy is measured as the averaged number in the last 30 periods over the 40-period run. Figure 16a shows that the slice rate increases and so the inference accuracy increases when the latency limit is relaxed. We then fix the latency set point at 80ms and change the power set point from 190W to 250W to test OptimML on TensorFlow. Figure 16b shows that the inference accuracy increases when the GPU server gets more power to use because the slice rate becomes higher. Those results are consistent with previous experimental results on PyTorch and demonstrate that OptimML can achieve the best inference accuracy under the desired power and latency limitations.

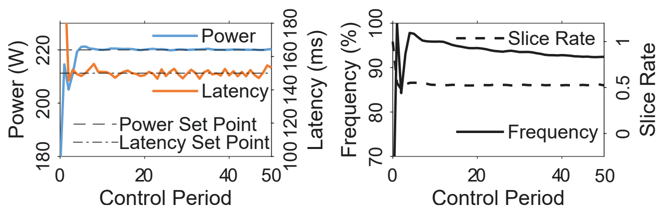
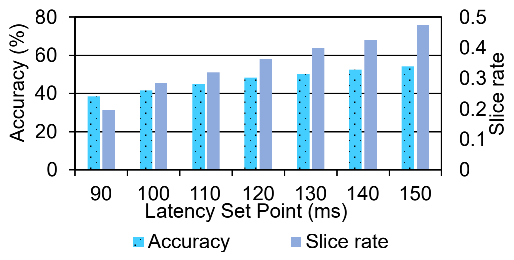
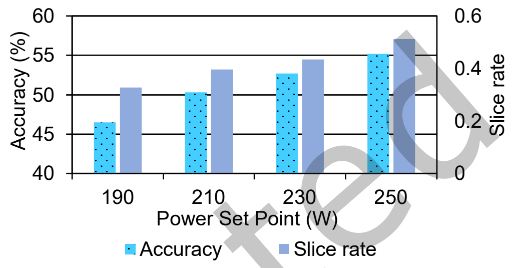


Fig. 17. Typical runs of OptimML on MXNet. Both power and latency are successfully controlled to their set points.



(a) Under different latency set points.



(b) Under different power set points.

Fig. 18. ML inference accuracy of OptimML on MXNet in the steady state.

MXNet is an ML framework that is designed by Apache mainly for the considerations of efficiency and flexibility. We now test OptimML on MXNet. In our experiments, the image batch size is 100. The set points for power and latency are 220W and 150ms, respectively. Figure 17 shows that both power and latency are promptly controlled to their respective set points. We then examine the inference accuracy by varying the power set points and latency set points within the same ranges used by TensorFlow above. When we test different power set points, the latency set point is fixed at 120ms. Similarly, the power set point is fixed at 200W when we test different latency set points. Figures 18a and 18b show that when the set point increases, the SLO and power limitations are more relaxed, respectively, so the inference accuracy improves as expected. To summarize this set of experiments, OptimML does not rely on a specific ML framework. For all the three tested ML frameworks, PyTorch, TensorFlow, and MXNet, OptimML performs consistently to meet both the power limitation and SLO, while achieving the highest ML inference accuracy.

8 CONCLUSION

Data center servers must enforce power capping to prevent power overload or overheating and to enable safe power oversubscription. In the meantime, they have to meet stringent SLO requirements such as the latency constraints of their ML applications. In this paper, we have presented OptimML, a MIMO control framework that jointly controls both inference latency and server power by flexibly adjusting ML model slice rate and GPU frequency. Our extensive results on a hardware testbed with widely adopted ML framework (including PyTorch, TensorFlow, and MXNet) and ML model ResNet show that OptimML can have better control accuracy than both the state-of-the-art baselines (Ad-Hoc and SISO) and a well-designed MIMO control solution (P5+P25), due to its MIMO control-theoretic designs. Furthermore, an adaptive control scheme with online model switching and estimation is designed to achieve analytic assurance of control accuracy and system stability, even in the face of significant workload/hardware variations. As a result, OptimML can achieve the best ML inference accuracy for popular ML applications under the same power and latency constraints.

REFERENCES

[1] Mohamed S Abdelfattah, Łukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D Lane. 2020. Best of both worlds: Automl codesign of a cnn and its hardware accelerator. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[2] Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Communications of the ACM, Vol. 54 No. 5, Pages 67-77* (2011).

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. Curran Associates, Inc., 1877–1901.

[4] Shaofeng Cai, Gang Chen, Beng Chin Ooi, and Jinyang Gao. 2019. Model Slicing for Supporting Complex Analytics with Elastic Inference Cost and Resource Constraints. In *Proceedings of the VLDB Endowment*, Vol. 13. 86–99. https://doi.org/10.14778/3364324.3364325

[5] Ming Chen, Xiaorui Wang, and Xue Li. 2011. Coordinating Processor and Main Memory for Efficient Server Power Control. In *Proceedings of the 25th International Conference on Supercomputing (ICS)*.

[6] Yu Cheng, Felix X. Yu, Rogerio S. Feris, Sanjiv Kumar, Alok Choudhary, and Shi-Fu Chang. 2015. An Exploration of Parameter Redundancy in Deep Networks with Circulant Projections. In *2015 IEEE International Conference on Computer Vision (ICCV)*. 2857–2865. https://doi.org/10.1109/ICCV.2015.327

[7] CIFAR 2021. The CIFAR-10 And CIFAR-100 Datasets Website. https://www.cs.toronto.edu/~kriz/cifar.html

[8] colab 2024. The Colab Website. https://colab.research.google.com/

[9] Emily L. Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 27. 1–9.

[10] Bruno Diniz, Dorgival Guedes, Wagner Meira, and Ricardo Bianchini. 2007. Limiting the Power Consumption of Main Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*.

[11] Peter Dorato, Vito Cerone, and Chaouki Abdallah. 1994. *Linear-Quadratic Control: An Introduction*. Simon and Schuster, Inc., New York, NY.

[12] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*.

[13] G.F. Franklin, J.D. Powell, and M. Workman. 1998. *Digital Control of Dynamic Systems*. Ellis-Kagle Press, Half Moon Bay, CA.

[14] Xing Fu, Xiaorui Wang, and Charles Lefurgy. 2011. How Much Power Oversubscription is Safe and Allowed in Data Centers?. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC)*.

[15] Ross Girshick. 2015. Fast R-CNN. In *2015 IEEE International Conference on Computer Vision (ICCV)*. 1440–1448. https://doi.org/10.1109/ICCV.2015.169

[16] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 199–213.

[17] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*.

[18] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[20] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. 2004. *Feedback Control of Computing Systems*. John Wiley and Sons, Inc., Hoboken, New Jersey.

[21] Sunpyo Hong and Hyesoon Kim. 2010. An Integrated GPU Power and Performance Model. In *2010 37th Annual International Symposium on Computer Architecture (ISCA)*. 280–289. https://doi.org/10.1145/1815961.1815998

[22] M. Horowitz, E. Alon, D. Patil, S. Naffziger, Rajesh Kumar, and K. Bernstein. 2005. Scaling, power, and the future of CMOS. In *2005 IEEE InternationalElectron Devices Meeting (IEDM)*. 7–15. https://doi.org/10.1109/IEDM.2005.1609253

[23] Hanzhang Hu, Debadeepta Dey, Martial Hebert, and J Andrew Bagnell. 2019. Learning anytime predictions in neural networks via adaptive loss balancing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3812–3821.

[24] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. Multi-Scale Dense Networks for Resource Efficient Image Classification. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Hk2aImxAb

[25] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2018. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research* 18, 187 (2018), 1–30.
[26] R. Kulhavý and M. Kárný. 1984. Tracking of Slowly Varying Parameters by Directional Forgetting. *IFAC Proceedings Volumes* 17, 2 (1984), 687–692.
[27] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4013–4021. https://doi.org/10.1109/CVPR.2016.435
[28] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. 2015. Speeding-up Convolutional Neural Networks Using Fine-Tuned Cp-Decomposition. In *2015 3rd International Conference on Learning Representations (ICLR)*. 1–11.
[29] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. 2007. Server-Level Power Control. In *2007 4th International Conference on Autonomic Computing (ICAC)*. 1–10. https://doi.org/10.1109/ICAC.2007.35
[30] En Li, Zhi Zhou, and Xu Chen. 2018. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*. 31–36.
[31] Jinhua Lin, Lin Ma, and Yu Yao. 2019. A Fourier domain acceleration framework for convolutional neural networks. *Neurocomputing* 364 (2019), 254–268. https://doi.org/10.1016/j.neucom.2019.06.080
[32] C. Liu, A. Sivasubramaniam, M. Kandemir, and M.J. Irwin. 2005. Exploiting barriers to optimize power consumption of CMPs. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1–10. https://doi.org/10.1109/IPDPS.2005.211
[33] Xue Liu, Xiaoyun Zhu, Pradeep Padala, Zhikui Wang, and Sharad Singhal. 2007. Optimal multivariate control for differentiated services on a shared hosting platform. In *2007 46th IEEE Conference on Decision and Control*. 3792–3799.
[34] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 301–312.
[35] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. 2012. GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures. In *2012 41st International Conference on Parallel Processing (ICPP)*. 48–57. https://doi.org/10.1109/ICPP.2012.31
[36] Arnab Neelim Mazumder, Jian Meng, Hasib-Al Rashid, Utteja Kallakuri, Xin Zhang, Jae-Sun Seo, and Tinoosh Mohsenin. 2021. A survey on the optimization of neural network accelerators for micro-ai on-device inference. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11, 4 (2021), 532–547.
[37] Model Slicing 2021. The Model Slicing Github Website. https://github.com/ooibc88/modelslicing
[38] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[39] Hanyu Peng, Jiaxiang Wu, Shifeng Chen, and Junzhou Huang. 2019. Collaborative Channel Pruning for Deep Networks. In *2019 36th Proceedings of International Conference on Machine Learning Research (PMLR)*, Vol. 97. 5113–5122.
[40] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. 2008. No power struggles: Coordinated multi-level power management for the data center. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[41] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey S. Chase. 2006. Ensemble-level Power Management for Dense Blade Servers.. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*.
[42] Manuele Rusci, Marco Fariselli, Alessandro Capotondi, and Luca Benini. 2020. Leveraging automated mixed-low-precision quantization for tiny edge microcontrollers. In *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning: Second International Workshop, IoT Streams 2020, and First International Workshop, ITEM 2020, Co-located with ECML/PKDD 2020, Ghent, Belgium, September 14-18, 2020, Revised Selected Papers 2*. Springer, 296–308.
[43] Mehmet Savasci, Ahmed Ali-Eldin, Johan Eker, Anders Robertsson, and Prashant Shenoy. 2023. DDPC: Automated Data-Driven Power-Performance Controller Design on-the-fly for Latency-sensitive Web Services. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 3067–3076.
[44] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=B1ckMDqlg
[45] Tensorflow Lite 2024. The TensorFlow Lite Website. https://www.tensorflow.org/lite
[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 30. Curran Associates, Inc.
[47] Haoyu Wang and Chengguang Ma. 2021. An optimization of im2col, an important method of CNNs, based on continuous address access. In *2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE)*. 314–320. https://doi.org/10.1109/ICCECE51280.2021.9342343
[48] Xiaorui Wang and Ming Chen. 2008. Cluster-level Feedback Power Control for Performance Optimization. In *Proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
[49] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W. Keller. 2009. SHIP: Scalable Hierarchical Power Control for Large-Scale Data Centers. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[50] Xiaorui Wang, Xing Fu, Xue Liu, and Zonghua Gu. 2009. Power-Aware CPU Utilization Control for Distributed Real-Time Systems. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[51] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. 2018. SkipNet: Learning Dynamic Routing in Convolutional Networks. In *2018 Proceedings of the European Conference on Computer Vision (ECCV)*. 409–424.

[52] Yue Wang, Tan Nguyen, Yang Zhao, Zhangyang Wang, Yingyan Lin, and Richard Baraniuk. 2018. EnergyNet: Energy-Efficient Dynamic Inference. In *2018 32nd Conference on Neural Information Processing Systems (NIPS)*. 1–5.

[53] Yefu Wang, Xiaorui Wang, Ming Chen, and Xiaoyun Zhu. 2008. Power-Efficient Response Time Guarantees for Virtualized Enterprise Servers. In *Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS)*. 303–312. https://doi.org/10.1109/RTSS.2008.20

[54] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6071–6079. https://doi.org/10.1109/CVPR.2017.643

[55] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Richard G. Baraniuk, and Yingyan Lin. 2020. Drawing Early-Bird Tickets: Towards More Efficient Training of Deep Networks. In *2020 8th International Conference on Learning Representations (ICLR)*. 1–13.

[56] P. C. YOUNG and J. C. WILLEMS. 1972. An approach to the linear multivariable servomechanism problem†. *Internat. J. Control* 15, 5 (1972), 961–979.

[57] Qianru Zhang, Meng Zhang, Tinghuan Chen, Zhifei Sun, Yuzhe Ma, and Bei Yu. 2019. Recent advances in convolutional neural network acceleration. *Neurocomputing* 323 (2019), 37–51. https://doi.org/10.1016/j.neucom.2018.09.038

Received 28 September 2023; revised 20 February 2024; accepted 22 April 2024