A Computing Task Allocation System and Method for Distributed Inference in Large Language Models

Released: 03/30/2024

Claims

1. This section describes a system for allocating computational tasks for distributed inference of large language models, suitable for Android phones. The features include:
An environment deployment module for setting up a distributed machine learning environment on mobile devices.
A model segmentation module for dividing large models on the PC side into smaller, executable ONNX sub-model files suitable for mobile devices.
A linear planning module to evaluate the computational and communication costs of sub-models, obtaining the best deployment solution.
A model execution module that deploys the model on mobile devices according to the linear planning solution, performing inference.

2. According to the system described in claim 1, its characteristic is that the linear planning module considers parameters including the model's FLOPs and the amount of parameters, the device's speed in processing FLOPs, the device's maximum memory, communication delay, bandwidth, jitter, and packet loss rate. The goal is to obtain a solution that minimizes inference time.

3. According to the system described in claim 2, its characteristic is that the goal of the linear planning optimization is the decoder layer of the model. The method used is mixed-integer linear planning, considering the variables as a two-dimensional binary model distribution matrix.

4. According to the system described in claim 2, its characteristic is that it models the packet loss rate with a binomial distribution as a mathematical penalty term, introducing nonlinear uncertainty, which aligns with real-world situations.

5. According to the system described in claim 1, its characteristic is that the model execution module uses the large language model's decoder layer as the server side, with the embedding layer and downstream tasks acting as the client side to perform inference in a closed-loop execution.

6. A method for allocating computational tasks for distributed inference of large language models, characterized by the following steps:

Step 1: Set up a distributed learning environment on mobile devices.
Step 2: Segment the large language model into several ONNX sub-models.
Step 3: Linearly plan model parameters and constrain model deployment:
For the several model layers segmented from the model, use the deployment binary matrix of the sub-models as variables. Model the minimum inference time expression based on variables such as the model's FLOPs, the device's processing speed for FLOPs, communication delay, bandwidth, jitter, and packet loss rate. Perform mixed-integer linear planning based on the device's maximum memory and the amount of the model's parameters to constrain the variables.
Step 4: Establish client and server sides on mobile devices and load the model for inference:
The client side loads the embedding and downstream task layers, controlling the model's input and output. The server side loads the decoder layer to complete pipeline-style inference computation.

7. The characteristic of the method for allocating computational tasks for distributed inference of large language models described in claim 6 lies in that, in step 3, the linear planning scheme satisfies:

$$minimize\ (T_{compute}+T_{data}+T_{complexity}+T_{QoS})$$

Where,

$$T_{complexity}=w_c\cdot\sum\sum x_{i,j}$$

$$T_{QoS}=\sum\sum w_{q1}\cdot Jitter_{i,j}+w_{q2}\cdot t_{i,j}\cdot PLR_{i,j}+w_{q3}\cdot PLR^2_{i,j}$$

$$T_{compute}=\sum\sum \epsilon_{i,j}\cdot x_{i,j}$$

$$T_{data}=\sum\sum (L_{i,j}+t_{i,j})$$

$$t_{i,j}=\frac{O_{i,j}}{E_p\cdot B_{i,j}}$$

$$\epsilon_{i,j}=\frac{NumFlop_{modj}}{FLOP/s_i}$$

Here, $T_{complexity}$ represents the system complexity penalty; T_{QoS} represents the link quality penalty; $Jitter, t, PLR$ respectively represent the jitter, transmission time, and packet loss rate for the corresponding link $\langle i,j\rangle$; i,j are the identifiers for communication nodes; $w_c, w_{q1}, w_{q2}, w_{q3}$ are weights for the corresponding terms, set according to system complexity and network quality requirements, with typical values being $w_c=1, w_1=10,\ w_2=1,\ w_3=10000$; E_p represents the effective payload efficiency of the communication protocol, with a typical value of 0.3; $O_{i,j}$ is the data volume output on the link $\langle i,j\rangle$; $B_{i,j}$ is the theoretical bandwidth of the link $\langle i,j\rangle$; $L_{i,j}$ is the delay on the link $\langle i,j\rangle$; $NumFlop_{modj}$ is the number of floating-point operations required by submodule j; $FLOP/s_i$ is the number of floating-point operations per second for device i; $x_{i,j}\in\{0,1\}$ represents the allocation of submodule j on device i.

8. According to the method described in claims 6 and 7, it is characterized in that step 3 should

also include explicit constraints:

$$\forall\ i \in \{0, \ldots, m-1\} \sum_{j=0}^{n-1} M_{mod_j} \cdot x_{i,j} \leq \beta \cdot M_{device_i}$$

and implicit constraints:

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} x_{i,j} = n$$

$$x[0,0] = 1$$

$$\forall\ j \in \{0, \ldots, n-1\} \sum_{i=0}^{m-1} x_{i,j} = 1$$

Where m and n respectively represent the number of devices and the number of models; M_{mod} and M_{device} respectively indicate the parameter amount of the model and the maximum memory of the device; β is the maximum proportion of memory that can be allocated to model inference by the device; $x_{i,j} \in \{0,1\}$ represents the allocation of submodule j on device i.

Full Text Of The Manual

Technology Field

This invention relates to a scheduling method for computational tasks, specifically designed for a computational task allocation system and method for distributed inference of large language models.

Technology Background

In prior research on inference using distributed edge devices, the optimization of distributed inference time focused on minimizing the sum of computation time and data transmission time. While this optimization is theoretically valid, the increased complexity of the system in practical deployments can lead to higher data transmission costs and risks.

Compared to computation, data transmission often takes more time. Transmission time needs to consider not only bandwidth and delay but also other factors affecting network communication quality, such as jitter and packet loss rate. Additionally, the actual effective payload of communication protocols is a significant issue when calculating transmission time.

To address the problem of excessive communication costs in distributed edge device inference for large language models, an optimized computational task allocation system and method have been developed. This system aims to balance computational and communication demands, taking into account the complexities and realities of networked environments to provide more efficient and reliable distributed inference.

Summary Of The Invention

The problem addressed by this invention is to propose a computational task allocation system and method for distributed inference of large language models. It aims to enable distributed operation of large language models for inference and other tasks on mobile devices by utilizing model splitting, pipelined inference, and linear programming, thereby fully leveraging edge computing power.

The technical solution is as follows:

The invention proposes a computational task allocation system for distributed inference of large language models, suitable for Android smartphones, characterized by the following components:

An environment deployment module for setting up a distributed machine learning environment on the smartphone. It installs a complete Linux Ubuntu system and a Conda Python virtual environment compatible with the ARM architecture.
A model segmentation module for splitting large models on the PC side into executable sub-model files for the smartphone. The splitting is done according to all layers defined in the model structure, and all are converted into the ONNX file format executable on the smartphone.

A linear programming module for evaluating the computational and communication costs of the sub-models to obtain the optimal deployment plan. It considers the model's FLOPs and parameter size, the device's processing speed for FLOPs, the device's maximum memory, communication latency, bandwidth, jitter, and packet loss rate. It models the packet loss rate using a binomial distribution as a mathematical penalty term to add non-linear uncertainty. The linear programming variable is a two-dimensional binary matrix, with the target being the decoder layer of the model. The method is mixed-integer linear programming, aiming to minimize the inference time to obtain the solution.

A model execution module for deploying the model on several smartphones according to the planned scheme. It separately launches the client and server loading codes for model inference. The sub-model inference code is written based on the ONNX runtime, with the model's decoder layer as the server and the embedding layer and downstream tasks as the client, executing inference in a closed loop.

The invention proposes a computational task allocation method for distributed inference of large language models:

Step 1: Set up a distributed learning environment on the smartphone:
Deploy the Termux programming environment on Android, in which a complete Linux Ubuntu system is set up. In the Ubuntu system, a Conda Python virtual environment is built, and basic Python libraries are downloaded.

Step 2: Segment the large language model into several sub-models:

Modify the model source code on Huggingface, adding model conversion code at the interfaces between layers. The goal is to transform the model from the Huggingface bin format into ONNX format sub-models executable on the smartphone. At the same time, the model's name, input, and output are explicitly converted. The input dimension needs to be set as dynamic to accommodate embedding discrete variables of different lengths.

Step 3: Linear programming for model parameters, constraining model deployment:
For the decoder layer split from the model, use the deployment binary matrix of the sub-model as the variable. Model the minimum inference time expression based on variables such as the model's FLOPs, device processing speed for FLOPs, communication latency, bandwidth, jitter, and packet loss rate. Constrain the variables based on the device's maximum memory and the model's parameter size, and perform mixed-integer linear programming. The linear programming scheme satisfies:

$$minimize\ \left(T_{compute}+T_{data}+T_{complexity}+T_{QoS}\right)$$

Where,

$$T_{complexity}=w_c\cdot\sum\sum x_{i,j}$$

$$T_{QoS}=\sum\sum w_{q1}\cdot Jitter_{i,j}+w_{q2}\cdot t_{i,j}\cdot PLR_{i,j}+w_{q3}\cdot PLR_{i,j}^2$$

$$T_{compute}=\sum\sum \epsilon_{i,j}\cdot x_{i,j}$$

$$T_{data}=\sum\sum\left(L_{i,j}+t_{i,j}\right)$$

$$t_{i,j}=\frac{O_{i,j}}{E_p\cdot B_{i,j}}$$

$$\epsilon_{i,j}=\frac{NumFlop_{modj}}{FLOP/s_i}$$

Here, $T_{complexity}$ represents the system complexity penalty; T_{QoS} represents the link quality penalty; $Jitter, t, PLR$ respectively represent the jitter, transmission time, and packet loss rate for the corresponding link $\langle i,j\rangle$; i,j are the identifiers for communication nodes; $w_c, w_{q1}, w_{q2}, w_{q3}$ are weights for the corresponding terms, set according to system complexity and network quality requirements, with typical values being $w_c=1, w_1=10,\ w_2=1,\ w_3=10000$; E_p represents the effective payload efficiency of the communication protocol, with a typical value of 0.3; $O_{i,j}$ is the data volume output on the link $\langle i,j\rangle$; $B_{i,j}$ is the theoretical bandwidth of the link $\langle i,j\rangle$; $L_{i,j}$ is the delay on the link $\langle i,j\rangle$; $NumFlop_{modj}$ is the number of floating-point operations required by submodule j; $FLOP/s_i$ is the number of floating-point operations per second for device i; $x_{i,j}\in\{0,1\}$ represents the allocation of submodule j on device i.

And explicit and implicit constraints:

$$\forall\ i \in \{0,\dots,m-1\} \sum_{j=0}^{n-1} M_{mod_j} \cdot x_{i,j} \leq \beta \cdot M_{device_i}$$

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} x_{i,j} = n$$

$$x[0,0] = 1$$

$$\forall\ j \in \{0,\dots,n-1\} \sum_{i=0}^{m-1} x_{i,j} = 1$$

Where m and n respectively represent the number of devices and the number of models; M_{mod} and M_{device} respectively indicate the parameter amount of the model and the maximum memory of the device; β is the maximum proportion of memory that can be allocated to model inference by the device; $x_{i,j} \in \{0,1\}$ represents the allocation of submodule j on device i.

Step 4: Set up the client and server on the smartphone, and load the model for inference:
The client loads the embedding and downstream task layers, controlling the model's input and output. The server loads the decoder layer, completing the pipeline-style inference computation.

This invention proposes a computational task allocation system and method for distributed inference of large language models. Due to its distributed characteristics and the ability to deploy optimally on edge devices such as smartphones for large language model inference, it has high development potential and application scenarios in the current context of GPU-based inference and training.

Embodiment

The system and method in question are explained in detail through an example, divided into four parts:

Part 1: Environment Deployment
We selected five Pixel 4 XL devices with Android 12 or above, and installed the Termux APK package via adb. After installing Termux, we updated the package list with `pkg update/upgrade`, and installed a complete Ubuntu system using `proot-distro install`. Entered the Ubuntu system as root to install front-end dependency packages such as wget and vim. Since Proot by default uses Python from Termux, we specified to temporarily use Python inside Proot with a command. Then, we downloaded the Miniforge installation script for ARM architecture using wget and completed the Conda setup by running the script. A Python 3.10 environment was created with `conda create`, and we entered this virtual environment to download ONNX, ONNX Runtime, and other related Python libraries via pip. This concludes the environment deployment.

Part 2: Model Segmentation
In this example, we used the Llama-7B large model, approximately 13GB in size. Huggingface integrates most models and provides a rich way of calling them, enabling us to obtain the Llama

model structure file, `modeling_llama.py`, from the `transformer` source code's models/llama folder. By examining the structure code, we identified that the LlamaModel class aggregates various layers, such as LlamaMLP and LlamaDecoderLayer. Therefore, in the `forward` function of this class, we defined different ONNX sub-models at the interfaces between layers. Taking the LlamaDecoderLayer as an example, its inputs are `hidden_states`, `attention_mask`, `position_ids`, `past_key_value`, `output_attentions`, and `use_cache`, and its outputs are `hidden_out`, `past_key`, and `past_value`. We imported the layer's input, output, and structure into the ONNX model using `torch.onnx.export`. Since invoking this model for inference requires handling input text of various dimensional lengths, some dimensions were dynamically assigned values based on the input text when defining the ONNX model. After defining all ONNX sub-models, we used the modified transformer source code library to perform model inference with `model.generate`, obtaining separate sub-model files. Note, models with downstream tasks required conversion of related layers in the `LlamaForCausalLM` class.

Part 3: Linear Programming

After the model segmentation described above, we obtained one embedding model, one LlamaRMSNorm model, one downstream task lm_head model, and 32 LlamaDecoder models. Considering that the embedding, LlamaRMSNorm, and lm_head models have smaller parameter sizes and are more oriented towards the client application layer, only the LlamaDecoder models are included in the linear programming scope.

For linear programming, we need a variable, an expression, and several constraints. The essence of the experiment is to optimally allocate these 32 models to four smartphone devices. Therefore, we can define a two-dimensional binary matrix X as the variable. The dimensions of this matrix are m×n, and it can only take values in $\{0,1\}$. Here, $x[i,j]$ represents the allocation of model j on device i. This way, the complete model allocation task can be uniformly represented by matrix X. The core of this study is to accelerate the inference of large language models by finding the minimum inference time, which is composed of computation time and communication time. Therefore, the expression can be the sum of computational and communication costs.

The calculation of the computational cost $T_{compute}$ is relatively straightforward. It can be obtained by summing the ratio of the FLOPs of all models to the processing speed of FLOPs of the devices they are allocated to. Formally, this can be expressed as:

$$T_{compute} = \sum \sum \epsilon_{i,j} \cdot x_{i,j}$$

$$\epsilon_{i,j} = \frac{NumFlop_{modj}}{FLOP/s_i}$$

Among them, $NumFlop_{modj}$ is the number of floating-point operations required for submodule j; $FLOP/s_i$ is the number of floating-point operations per second for device i.

The Fvcore machine learning library offers a means to calculate the FLOPs of a model. By invoking fvcore.nn.FlopCountAnalysis(model) at the previous model splitting point, one can obtain the FLOPs of each layer. In addition, a Python script is run on each device, defining a simple convolutional neural network. Through the FlopsBenchmarker class in the torch.utils.benchmark module provided by the PyTorch library, the benchmarker.benchmark function is called to measure the device's FLOPs (floating-point operations per second) speed.

The calculation of communication cost is relatively more complex. To better simulate the complexity of communication in real-world environments, four communication metrics are proposed: latency, bandwidth, jitter, and packet loss rate.

Delay and bandwidth are conventional quantities of communication, represented by T_{data}. T_{data} reflects the theoretical communication time. Formally, this can be expressed as:

$$T_{data} = \sum \sum (L_{i,j} + t_{i,j})$$

$$t_{i,j} = \frac{O_{i,j}}{E_p \cdot B_{i,j}}$$

Among them, $O_{i,j}$ represents the volume of output data to be transmitted over the link between devices i and j; $B_{i,j}$ denotes the theoretical bandwidth of the link between devices i and j; $L_{i,j}$ is the latency on the link between devices i and j; E_p stands for the effective payload efficiency of the communication protocol being used, with a typical value of 0.3.

The calculation of latency and bandwidth can be achieved by establishing a simple client-server communication between two devices. By sending a fixed amount of data (e.g., 1MB) and measuring the time taken, latency can be determined. The bandwidth can then be calculated as the ratio of the number of bytes to the latency. The output size of a model can be directly calculated based on its dimensions; for example, the total number of bytes for an output with dimensions [1,1,4096] in float32 format would be 114096*4 = 16,384 bytes.

Jitter and packet loss rate are considered as penalty terms in communication, represented by T_{QoS} for link quality. Jitter is defined as the difference between the maximum and minimum latency of data transmission. Packet loss rate, a random failure in wireless network communication, can be assessed using a binomial distribution. The mathematical expectation of the binomial distribution can be interpreted as the average number of transmission failures, such as packet loss, occurring within a transmission period, while the variance can be used to evaluate the uncertainty of the prediction. T_{QoS} can be represented as:

$$T_{QoS} = \sum \sum w_{q1} \cdot Jitter_{i,j} + w_{q2} \cdot t_{i,j} \cdot PLR_{i,j} + w_{q3} \cdot PLR_{i,j}^2$$

Here, $Jitter, t, PLR$ respectively represent the jitter, transmission time, and packet loss rate for the corresponding link $\langle i,j \rangle$; i,j are the identifiers for communication nodes; $w_c, w_{q1}, w_{q2}, w_{q3}$ are weights for the corresponding terms, set according to system complexity and network quality requirements, with typical values being $w_c = 1, w_1 = 10$, $w_2 = 1$, $w_3 = 10000$.

To simulate complex network conditions such as jitter and packet loss in a normal local area network, we can leverage the Netem network module in the Linux kernel to create a weak network environment. By installing the traffic control tool TC (Traffic Control) in the Ubuntu system, we can use commands like `tc qdisc add dev eth0 root netem loss 10%` and `tc qdisc add dev eth0 root netem delay 100ms 20ms` to increase packet loss rate and network jitter, respectively.

In general, the expression of linear programming is:

$$minimize\ \left(T_{compute}+T_{data}+T_{complexity}+T_{QoS}\right)$$

Among them, $T_{complexity}$ represents the system complexity penalty; w_c are weights for the corresponding terms, with typical values being $w_c=1$:

$$T_{complexity}=w_c\cdot\sum\sum x_{i,j}$$

Regarding the constraint terms of linear programming, considering the maximum memory of the device and the number of model parameters, define explicit constraint terms as:

$$\forall\ i\in\{0,\ldots,m-1\}\sum_{j=0}^{n-1}M_{mod_j}\cdot x_{i,j}\leq\beta\cdot M_{device_i}$$

Where m and n respectively represent the number of devices and the number of models; M_{mod} and M_{device} respectively indicate the parameter amount of the model and the maximum memory of the device; β is the maximum proportion of memory that can be allocated to model inference by the device; $x_{i,j}\in\{0,1\}$ represents the allocation of submodule j on device i.

The purpose of this constraint is to ensure that the total memory usage of all models loaded on a device does not exceed the given maximum memory capacity of the device. Additionally, considering the deployability of variables in real scenarios and the pipeline mode of running inference, there are the following implicit constraints:

$$\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}x_{i,j}=n$$

$$x[0,0]=1$$

$$\forall\ j\in\{0,\ldots,n-1\}\sum_{i=0}^{m-1}x_{i,j}=1$$

The purpose of these constraints is to ensure the complete deployment of all models across different devices, with the first decoder model always located in the first device, and to prevent any model from being deployed more than once.

After completing the modeling for linear programming, it is necessary to invoke an actual optimizer to perform the linear programming calculation. Given that this linear programming falls under the category of mixed-integer linear programming, general linear programming libraries are not suitable. Therefore, we utilize the independent third-party optimizer Gurobi, which not only supports continuous and mixed-integer linear problems but also provides good support for matrix operations. The following code block is used to define the linear programming task and variables:

```
import gurobipy as gp
```

```
from gurobipy import GRB
model = gp.Model("milp")
model.Params.NumericFocus = 3
x = model.addMVar((m,n), vtype=GRB.BINARY, name="x")
```

The mathematical expressions are converted into code, and the objective is set using `model.setObjective(Tcompute + Tdata + Tcomplexity + TQoS, GRB.MINIMIZE)`. Subsequently, constraints are established, and the optimization process begins:

```
for i in range(m):
    model.addConstr(gp.quicksum(Mmod[j]*x[i,j] for j in range(n))<=  beta*Mdevice[i])
model.addConstr(gp.quicksum(x[i,j] for j in range(n) for i in range(m)) == n)
for j in range(n):
    model.addConstr(gp.quicksum(x[i,j] for i in range(m)) == 1)
#  Limit the value of x[i,j]
model.addConstr(x[0,0] == 1)
model.optimize()
```

Part 4: Model Execution

After importing the models into each device via `adb` commands, Python code needs to be written to execute on the devices. The smartphones are divided into two categories: clients and servers. The client loads the embedding, norm, and downstream task head modules, responsible for inputting text and converting the output tensor into text token information. The server loads several decoder modules. To optimize inference speed during code execution, we added code to store the Key and Value values of the attention layer to avoid redundant matrix dimension calculations for subsequent tokens. With this design, when loading input text into the model, only the current token needs to be loaded after the initial loading of all tokens. To run ONNX model code, `onnxruntime.InferenceSession(onnxfile)` is called to load the model into memory, and `ort.InferenceSession(onnxfile).run` is used to infer the model.

This study presents a computational task allocation system and method for distributed inference of large language models. By using a universally applicable optimization method, it allows edge computing power to participate in the application of large language models, which is beneficial for the development of edge computing. Although detailed descriptions of the embodiments of the invention have been provided in the previous text, it should be understood that modifications and variations can be made by those skilled in the art without departing from the scope and spirit of the invention as disclosed in the claims. Moreover, it should be noted that other embodiments of the invention may exist and can be realized through various means.

Appendix

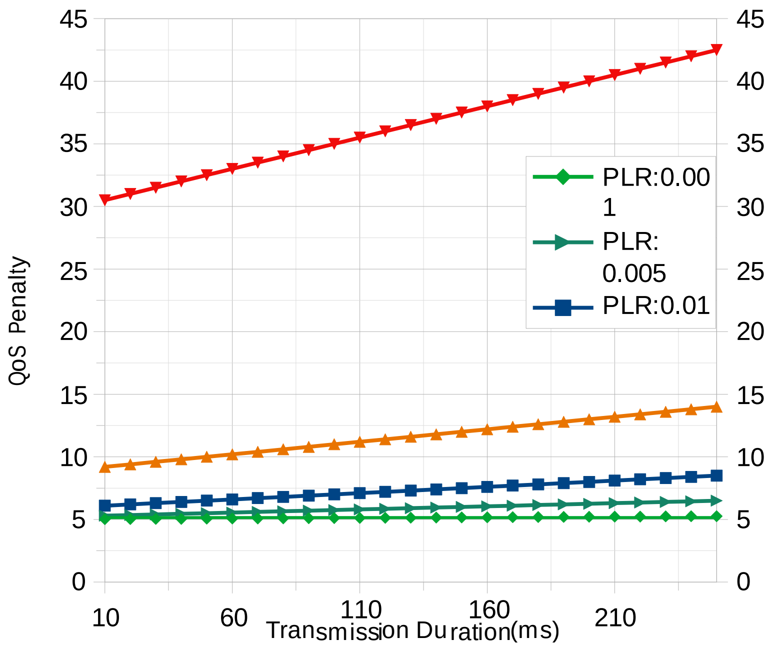

Figure 1. PLR impact on QoS Penalty

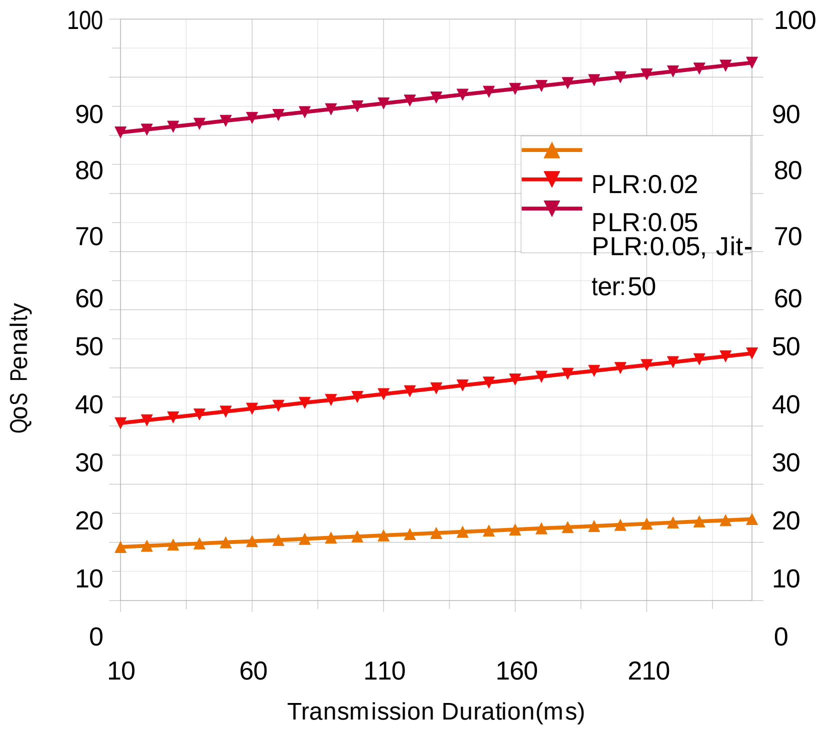

Figure 2. Jitter Impact on QoS Penalty