TOC and LSTM Based Resource Scheduling

Nipun S Nair
VIT Vellore

Anamika
VIT Vellore

Vishnu P K
VIT Vellore

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et.

1 Literature Review

Chang et al. [1] applied a TOC-Based approach to address memory allocation in cloud storage, which uses market information to build a rolling forecast. This demonstrates that TOC principles can be extended to resource scheduling problems in dynamic environments.

2 Methodology

This section describes the procedures, tools, and methods used to conduct the research.

2.1 Training LSTM Model to predict bottlenecks

2.1.1 Synthetic Data Generation

We simulate a realistic time-series dataset representing server resource usage over time. The goal is to generate data that mimics real-world system behavior to train an LSTM model for bottleneck prediction.

Parameter	**Description**
NUM_SERVERS	Number of simulated servers
SIM_TIME	Total simulated duration (in time units)
TASK_INTERVAL	Mean inter-arrival time for new tasks
CPU_CAPACITY	Max CPU Usage (100%)
NET_CAPACITY	Max network bandwidth (100%)

Table 1: Simulation Parameters

2.1.1.1 Poisson-Based Task Arrival

Task arrival times are based on a Poisson process, simulated using the exponential distribution:

$$\text{Interarrival Time} \sim \text{Exponential}\left(\lambda = \frac{1}{\text{TASK\_INTERVAL}}\right)$$

This introduces realistic irregularity in task arrivals—mimicking user requests, job submissions, or packet arrivals.

2.1.1.2 Time-Varying Resource Usage Patterns

We simulate periodic system load using sinusoidal functions to represent diurnal load patterns, cyclic CPU spikes, and network traffic fluctuations.

CPU Usage

$$\text{cpu\_base}(t) = 50 + 40 * \sin\left(\frac{2\pi t}{\text{PERIOD}}\right)$$

Network In/Out

$$\text{net\_base}(t, s) = 30 + 25 * \sin\left(\frac{2\pi(t + s * 10)}{\text{PERIOD}}\right)$$

Where:

- t: current timestamp
- s: server index (adds phase shift between servers)
- `PERIOD`: defines workload cycle length (e.g. peaks every 100 time units)

2.1.1.3 Gaussian Noise for Realism

We inject Gaussian noise to simulate sensor or monitoring variability. This produces "wobble" on top of clean trends, similar to real monitoring data.

$$\text{cpu} = \text{clip}(\text{cpu\_base} + \mathcal{N}(0, 10), 0, 100)$$

$$\text{net\_in}, \text{net\_out} = \text{clip}(\text{net\_base} + \mathcal{N}(0, 5), 0, 100)$$

2.1.1.4 Scheduled Bottlenecks

At specific intervals (e.g., for $t = 250$ to 260 on Server 1), we introduce high-load conditions to ensure predictable bottlenecks for model learning and testing.

- CPU, Net In, and Net Out are all set to $\geq 90\%$.

2.1.1.5 Bottleneck Label Definition

A binary bottleneck label is assigned based on an 80% utilization threshold:

$$\text{bottleneck} = \begin{cases} 1 \\ \text{if CPU} \geq 80 \quad \text{or Net In/Out} \geq 80 \\ 0 \\ \text{otherwise} \end{cases}$$

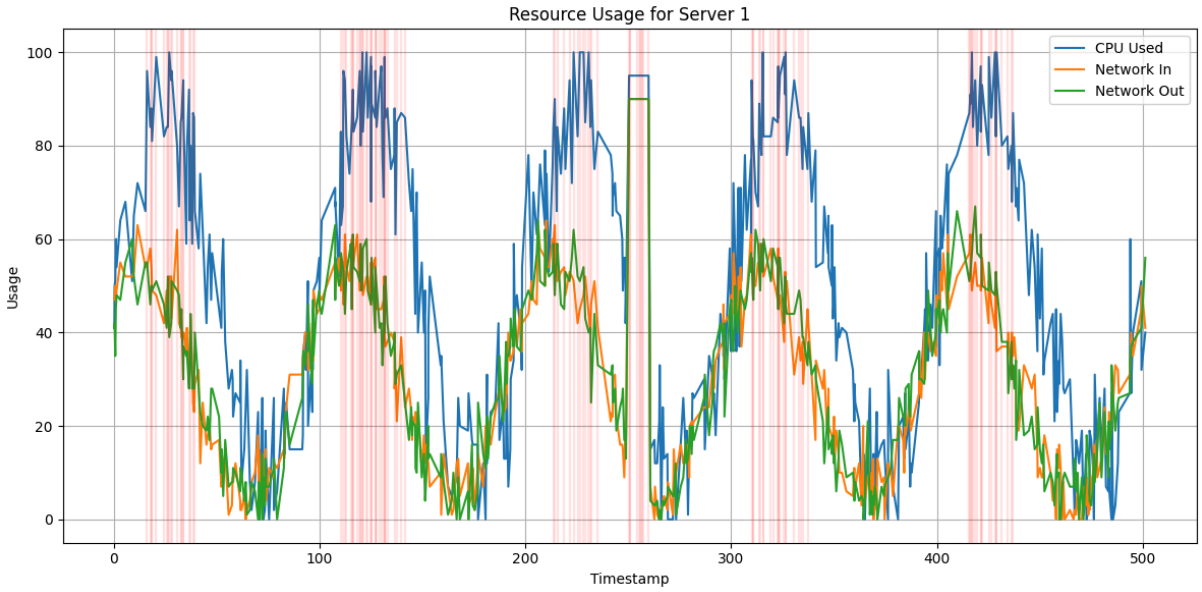

Figure 1: Synthetic Data - Compute and Network Resources

2.1.2 Training Model

To identify potential bottlenecks, we trained a Long-Short-Term Memory neural network, as it is well suited for learning patterns and dependencies in time series data.

To capture temporal patterns, the continuous time series data for each server was transformed into overlapping sequences of windows of size 20 timesteps, with each window serving as an input sequence and the following timestep's bottleneck status as the label (1/0). This information allows the model to learn the patterns leading up to a bottleneck.

The dataset was then split into training and test sets and a MinMax scaler was fit only on the training data and applied to both the sets. Finally we trained a recurrent neural network with an LSTM layer followed by dense layers optimized using binary cross-entropy loss, to classify whether a bottleneck would occur in the next timestep, given the previous 20 timesteps.

2.1.3 Evaluation

The model was evaluated on a test set of 293 samples, achieving an overall accuracy of 89%. The detailed performance metrics from the classification report are presented below.

	Precision	Recall	F1-Score	Support
No Bottleneck (0)	0.94	0.91	0.93	235
Bottleneck (1)	0.69	0.78	0.73	58
Accuracy			0.89	293
Macro Avg	0.82	0.85	0.83	293
Weighted Avg	0.89	0.89	0.89	293

Table 2: Classification Report

	Predicted: No Bottleneck	Predicted: Bottleneck
Actual: No Bottleneck	215	20
Actual: Bottleneck	13	45

Table 3: Confusion matrix of model predictions.

2.2 Algorithm 1 (Base) - Round Robin Simulation

To evaluate the performance of the advanced scheduling algorithms, a robust baseline is required. For this purpose, we implement a resource-aware variant of the classic Round Robin (RR) scheduling algorithm.

2.2.1 Theoretical Foundation

The core principle of Round Robin is to achieve fairness and prevent starvation by distributing tasks in a simple, cyclical sequence. Unlike more complex load-balancing algorithms that analyze system state to find the optimal server, RR employs a stateless, turn-based approach. It maintains a pointer to the last server that received a task and assigns the next incoming task to the subsequent server in the sequence.

Our implementation enhances this basic model by making it **resource-aware**. A pure RR scheduler would assign a task to the next server regardless of its state, which could lead to queue overloads and task failures if the server lacks the capacity. Our algorithm mitigates this by performing two critical checks before assignment:

1. **Queue Capacity Check:** The server's task queue must not be full.
2. **Resource Availability Check:** The server must have sufficient free CPU and network capacity to begin processing the specific task at that moment.

If the designated server fails these checks, the algorithm continues its cyclical search until an adequate server is found. If no server in the system can accept the task, it is rejected.

2.2.2 Algorithm Description

The scheduling process for each new incoming task is executed as follows:

1. **Initialization:** Upon the arrival of a new task, $\mathcal{J}$, the algorithm identifies the server that last received a task, denoted by its index I_{last}. The search for a suitable server begins at the next server in the sequence, $s_{I_{\text{start}}}$, where $I_{\text{start}} = (I_{\text{last}} + 1) \bmod N$.

2. **Cyclical Search:** The algorithm iterates through all active servers s_k in a circular order, starting from $s_{I_{\text{start}}}$.
3. **Eligibility Check:** For each candidate server s_k, it performs two validation steps:
   - It first verifies that the server's local task queue has not reached its maximum capacity, $Q_{\max}$.
   - If the queue has space, it then verifies that the server's available resources (CPU, network input, network output) are greater than or equal to the demands of task $\mathcal{J}$.
4. **Assignment or Rejection:**
   - The first server, s_k, that satisfies both conditions is selected as the target, S_{target}. The index I_{last} is updated to k, and the search terminates.
   - If the algorithm completes a full cycle and no server satisfies both conditions, the task $\mathcal{J}$ is rejected, constituting an SLA violation.

2.2.3 Mathematical Formulation

Let S be the set of N servers, indexed as $S = \{s_0, s_1, ..., s_{N-1}\}$. Let I_{last} be the index of the server that received the previous task.

A new task, J_{new}, arrives with resource demands $D = \{D_{\text{cpu}}, D_{\text{net\_in}}, D_{\text{net\_out}}\}$.

For any server s_i at time t, we define its state by:
- $Q_{s_i}(t)$: The current length of its task queue.
- $C_{s_i,\,\text{avail}}^{\text{cpu}(t)}$: The available CPU capacity.
- $C_{s_i,\,\text{avail}}^{\text{net\_in}(t)}$: The available network input capacity.
- $C_{s_i,\,\text{avail}}^{\text{net\_out}(t)}$: The available network output capacity.

Two boolean conditions must be met for a server s_i to be eligible to accept J_{new}:

1. **The Queue Capacity Condition**, $\text{Accept}_{Q(s_i)}$:

$$\text{Accept}_{Q(s_i)} = \left[Q_{s_i}(t) < Q_{\max}\right]$$

   where [.] is the Iverson bracket.

2. **The Resource Availability Condition**, $\text{Accept}_{R(s_i, J_{\text{new}})}$:

$$\text{Accept}_{R(s_i, J_{\text{new}})} = \left[D_{\text{cpu}} \leq C_{s_i,\,\text{avail}}^{\text{cpu}(t)}\right] \wedge \left[D_{\text{net\_in}} \leq C_{s_i,\,\text{avail}}^{\text{net\_in}(t)}\right] \wedge \left[D_{\text{net\_out}} \leq C_{s_i,\,\text{avail}}^{\text{net\_out}(t)}\right]$$

The algorithm searches for a target server, S_{target}, within the ordered sequence of candidate indices k, where k cycles from $(I_{\text{last}} + 1) \bmod N$ for N steps. The first server s_k for which $\text{Accept}_{Q(s_k)} \wedge \text{Accept}_{R(s_k, J_{\text{new}})}$ evaluates to true is chosen:

$$S_{\text{target}} = s_k$$

If no such server is found after checking all N servers, the task is rejected. This simple but robust algorithm provides an effective baseline for evaluating more complex scheduling strategies.

2.3 Algorithm 2 - Theory of Constraints with Fitness Calculation

This algorithm provides a practical implementation of TOC's **Drum-Buffer-Rope (DBR)** methodology for a parallel, dynamic server environment.

- **The Drum:** The system's identified constraint, whose processing rate dictates the optimal rate at which new work should be introduced.
- **The Buffer:** A small, managed queue of tasks placed before each resource. The buffer in front of the constraint is the most critical, as it must ensure the constraint is never idle due to a lack of work.
- **The Rope:** A signaling mechanism that links the constraint's buffer back to the system's entry point. It authorizes the release of new work into the system only when the constraint has the capacity to process it, thereby synchronizing the entire system to the pace of its slowest part.

The algorithm is composed of four primary components that map to the Five Focusing Steps of TOC: Constraint Identification, Flow Control (Dispatcher), Task Assignment, and System Scaling.

2.3.1 The Algorithm Components

Let S be the set of all servers. At any time t, the set of active servers is denoted by $S_{\text{active}(t)} \subseteq S$. Each server $s \in S$ has a maximum CPU capacity $C_{s,\text{cpu}}$ and network capacity $C_{s,\text{net}}$.

2.3.1.1 Component 1: Constraint Identification (Identify)

The first step is to dynamically and continuously identify the system's primary constraint. Instantaneous resource utilization is often volatile; therefore, a smoothing function is required to identify the most persistently loaded resource. We employ an **Exponentially Weighted Moving Average (EWMA)** for this purpose.

Let $U_{s,r}(t)$ be the instantaneous utilization of a resource $r \in \{\text{cpu}, \text{net}\}$ on a server s at time t. The utilization is a normalized value where $0 \leq U \leq 1$.

The smoothed utilization, $U.|_{s,r}(t)$, is calculated recursively:

$$U.|_{s,r}(t) = \big(\alpha \cdot U_{s,r}(t)\big) + \big((1-\alpha) \cdot U.|_{s,r}(t-\Delta t)\big)$$

Where:

- α is the smoothing factor ($0 < \alpha < 1$). A lower α results in a smoother, less volatile trendline.
- Δt is the time interval between measurements.

The system constraint at time t, denoted $C(t)$, is the specific resource (s, r) with the highest smoothed utilization across all active servers.

$$C(t) = \text{argmax}_{s \in S_{\text{active}(t)}, r \in \{\text{cpu, net}\}} \big\{U.|_{s,r}(t)\big\}$$

This identification process runs at a fixed interval, `CONSTRAINT_CHECK_INTERVAL`, to adapt to changing system loads.

2.3.1.2 Component 2: Flow Control via Dispatcher (Exploit & Subordinate)

The core of the DBR implementation is a centralized **Dispatcher** that acts as the "Rope." It manages a central priority queue of incoming tasks, B_{central}, and only releases work into the system based on the state of the identified constraint, $C(t)$.

Let $C_{s(t)}$ be the server component of the constraint $C(t)$. Let $Q_{s(t)}$ be the length of the local buffer (queue) of server s at time t, and let $Q_{\max}$ be the maximum configured size of this buffer.

The **Rope Condition**, Release(t), is a boolean function that determines if a new task should be released from B_{central}:

$$\text{Release}(t) = \left[Q_{C_{s(t)}}(t) < Q_{\max} \right]$$

This condition ensures that a new task is only introduced into the system when the constraint's buffer has capacity. This prevents the accumulation of excessive Work-in-Process (WIP) and paces the entire system to its bottleneck. If Release(t) is false, no tasks are dispatched, and they remain in the managed, prioritized central buffer.

2.3.1.3 Component 3: Task Assignment (Subordinate)

When the Rope Condition Release(t) is met, the highest-priority task, J_{next}, is selected from the central buffer:

$$J_{\text{next}} = \text{argmin}_{j \in B_{\text{central}}} \{P(j)\}$$

where $P(j)$ is the priority value of task j (lower is higher).

This task must then be assigned to an active server. This is a subordinate decision, designed to efficiently utilize non-constraint resources without disturbing the system's flow. The target server, S_{target}, is selected from the set of available servers, $S_{\text{avail}(t)}$, by finding the server with the smallest local buffer.

The set of available servers is defined as:

$$S_{\text{avail}(t)} = \left\{ s \in S_{\text{active}(t)} \mid Q_{s(t)} < Q_{\max} \right\}$$

The target server is then chosen by:

$$S_{\text{target}} = \text{argmin}_{s \in S_{\text{avail}(t)}} \left\{ Q_{s(t)} \right\}$$

This ensures the released task is routed to the most idle part of the system, minimizing its local wait time and keeping non-constraint resources productive.

2.3.1.4 Component 4: System Scaling (Elevate)

The final component is the autoscaler, which implements the "Elevate the Constraint" step. It modifies the size of the active server set, $\left|S_{\text{active}(t)}\right|$.

Let θ_{up} be the scale-up threshold and θ_{down} be the scale-down threshold. Let $N(t) = \left|S_{\text{active}(t)}\right|$ be the number of active servers.

Scale-Up Condition: The decision to scale up is based solely on the status of the constraint. If the smoothed utilization of the constraint resource exceeds the threshold, a new server is activated.

$$\text{ScaleUp}(t) = \left[U.|_{C(t)}(t) > \theta_{\text{up}}\right] \wedge [N(t) < N_{\max}]$$

This ensures that capacity is added precisely where it is needed to relieve the system's bottleneck.

Scale-Down Condition: The decision to scale down is based on overall system idleness. Let $U.|_{\text{sys}(t)}$ be the average CPU utilization across all active servers. To prevent premature scaling during the initial warm-up phase, a time condition T_{warmup} is included.

$$\text{ScaleDown}(t) = \left[t > T_{\text{warmup}}\right] \wedge \left[U.|_{\text{sys}(t)} < \theta_{\text{down}}\right] \wedge [N(t) > N_{\min}]$$

This allows the system to conserve resources when the overall demand is low, without being triggered by the intentionally low utilization of non-constraint servers during periods of high load.

2.4 Algorithm 3 - Theory of Constraints with LSTM Bottleneck Prediction

References

[1] K.-H. Chang, Y.-C. Chang, and Y.-S. Chang, "Applying theory of constraints-based approach to solve memory allocation of cloud storage," *International Journal of Systems Science: Operations & Logistics*, vol. 4, pp. 311–329, 2017, [Online]. Available: https://api.semanticscholar.org/CorpusID:113444483