Journal of Electrical and Computer Engineering Innovations
(JECEI)
Journal homepage: http://www.jecei.sru.ac.ir

Research paper

Early-Stage Resource-Bound Prediction for Threads Using Real-Time Kernel Event Analysis

Morteza Noferesti [*], ***Farzad Amiri Delouei***, ***Sarah Aryan***

Department of Engineering, Bozorgmehr University of Qaenat, Qaen, South Khorasan, Iran.

Article Info

Article History:
Received 20 January 2026
Reviewed 27 February 2026
Revised 28 May 2026
Accepted 11 June 2026

Keywords:
Performance analysis
Proactive scheduling
Kernel events
Behavioral profiling
Applied AI

[*]Corresponding Author's Email Address:
mnoferesti@buqaen.ac.ir

Abstract

Background and Objectives: Modern operating systems struggle to manage threads with dynamic resource demands, as traditional schedulers rely on reactive heuristics that often misclassify thread behavior. This paper introduces a proactive thread classification methodology that predicts resource-bound categories by analyzing kernel event streams in real time.

Methods: Our proposed five-step pipeline includes: (1) kernel event collection using LTTng, (2) system call categorization into a seven-category taxonomy covering 57 system calls, (3) PID/TID labeling based on resource usage, (4) feature extraction from the first five events, and (5) predictive modeling with multiple machine learning classifiers.

Results: Our evaluation of six machine learning models, including Random Forest, LightGBM, Stacked Ensemble, MLP, CNN-BiLSTM, and BERT shows that Random Forest delivers the optimal balance of high predictive performance (93.4% precision, 92.5% recall) and low inference latency (178 µs), outperforming both other ensemble methods and computationally expensive deep learning architectures. When applied to a real-world dataset, the approach achieves 89% precision in thread classification, which directly translates to significant system-level improvements: a 41% reduction in tail latency for interactive applications and sustained 93% CPU utilization for cpu-bound tasks.

Conclusion: This paper demonstrates the efficacy of a novel, proactive thread classification methodology that accurately predicts a thread's future resource-bound category within a critically short 100 µs window from its execution start. By instrumenting a five-step pipeline, the approach successfully translates fine-grained system call sequences into predictive signatures for resource constraints, such as identifying I/O-bound threads from read/write patterns. This early detection capability provides a timely and actionable foundation for operating system schedulers to preemptively optimize thread prioritization and resource allocation, thereby enhancing overall system performance and responsiveness.

This work is distributed under the CC BY license (http://creativecommons.org/licenses/by/4.0/)

How to cite this paper:
M. Noferesti, F. Amiri Delouei, S. Aryan, "Early-Stage resource-bound prediction for threads using real-time kernel event analysis," J. Electr. Comput. Eng. Innovations, 14(2): 583-597, 2026.

DOI: 10.22061/jecei.2026.12900.914

URL: https://jecei.sru.ac.ir/article_12573.html

2345- 3044/© 2026 The Authors. Published by Shahid Rajaee Teacher Training University.

Introduction

Efficient execution of multithreaded workloads in modern computing systems depends on accurately identifying and managing a thread's resource constraints, whether CPU, memory, or I/O-bound [1]. Unpredictable thread behavior, such as abrupt shifts between compute-intensive and latency-sensitive phases, degrades system performance, causing increased scheduling latency [2] and suboptimal hardware utilization [3]. Monitoring thread behavior is particularly essential for maintaining throughput and fairness [4].

Current operating systems rely on reactive scheduling strategies, using historical metrics like CPU usage or sleep durations to infer thread behavior [5]. These approaches struggle to adapt to dynamic workloads, frequently leading to misclassification, such as mistaking a CPU-bound thread for an I/O-bound one, or delayed adjustments that worsen resource contention [6], [7]. This challenge is particularly evident in heterogeneous workloads, where threads rapidly switch between resource-intensive and idle states.

The traditional scheduling hierarchy, represented in Fig. 1, comprises *long-term*, *medium-term*, and *short-term* schedulers that operate reactively. Processes transition through the job pool, memory, and CPU based on static policies or historical behavior, often resulting in suboptimal resource allocation [1]. Several kernel events occur within the OS to manage the process lifecycle [7]. For instance, *block_rq_insert* indicates a process being swapped out to disk by the medium-term scheduler, *io_event_irq* signals an I/O completion interrupt to the CPU, and *sched_swi*tch records the moment the short-term scheduler performs a context switch, selecting a new thread from the ready queue to execute on the CPU. The *short-term* scheduler makes these selections without predicting future resource demands, while the *medium-term* swaps processes only after memory pressure emerges [8]. This reactive approach struggles with dynamic workloads, particularly for threads that alternate between CPU-intensive and I/O-intensive phases, as the scheduling occurs only after conditions have changed rather than proactively anticipating workload shifts.

Proactive scheduling approaches improve traditional schedulers by anticipating thread resource demands before they affect system performance [9], [10]. Unlike reactive methods, these approaches classify threads as CPU-bound, I/O-bound, or memory-bound early in their lifecycle, often upon entering the job pool [1]. This allows the long-term scheduler to prioritize threads that optimize resource usage, such as combining I/O-bound and CPU-bound threads to enhance parallelism. The medium-term scheduler benefits by proactively identifying memory-bound threads, reducing unnecessary swapping through preemptive management of memory-intensive threads [11].

Critically, the short-term scheduler uses the following predictions:

- Prioritize I/O-bound threads before they block, minimizing latency,
- Allocate longer times to CPU-bound threads, reducing context switches, and
- Group threads with complementary resource demands (pairing CPU-bound and I/O-bound threads to maximize CPU and disk utilization).

By analyzing fine-grained kernel events, such as system call patterns, context switch frequencies, and memory pressure signals, proactive scheduling approaches can predict thread resource demands before bottlenecks occur [9]. This predictive capability enables schedulers to:

1. Preemptively prioritize latency-sensitive threads without starving CPU-bound tasks,
2. Reduce unnecessary context switches by accurately classifying threads early in their execution, and
3. Dynamically optimize resource allocation to align hardware utilization with actual thread demands.

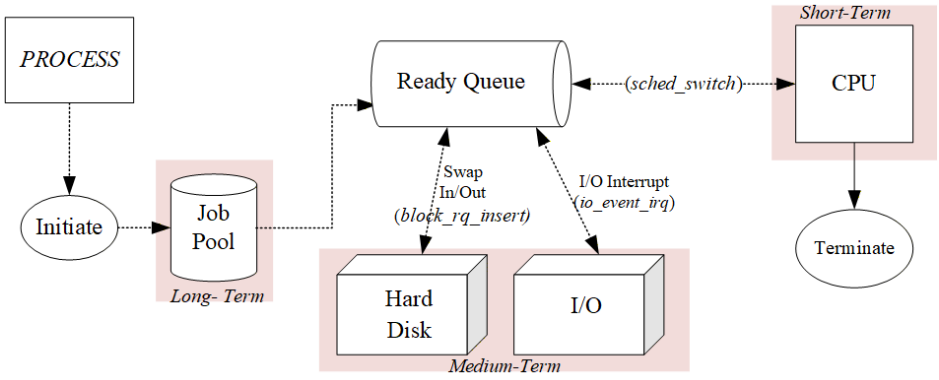

Fig. 1: Traditional reactive scheduling diagram.

This paper introduces a novel methodology for early detection of resource-bound threads through behavioral analysis of kernel event streams. Unlike conventional schedulers that rely on reactive metrics, such as Linux sleep averages or historical CPU usage, our approach examines low-level execution signatures, system call sequences, context switch patterns, and I/O event correlations to predict thread behavior before resource contention emerges. The proposed system combines (1) lightweight kernel tracing using LTTng, (2) a seven-category taxonomy of resource-bound indicators derived from system calls and scheduler events, and (3) machine learning models that map early execution patterns (as few as five events) to resource-bound categories. By converting kernel events into real-time behavioral fingerprints, our method addresses the latency/compute trade-off in modern systems, preventing delays from I/O-bound threads stalling CPU-intensive workloads and vice versa.

The predictive power of this methodology arises from multimodal event analysis: scheduling events (*sched_switch* frequency) indicate CPU contention, interrupt timings reveal hardware interaction costs, and I/O call sequences (*block_rq_complete* latencies) highlight emerging storage bottlenecks. Unlike statistical methods requiring prolonged observation, our lightweight feature extraction pipeline identifies these markers during thread initialization, enabling schedulers to preemptively adjust priorities, allocate resources, or initiate migrations. Experimental results demonstrate 89.2% prediction accuracy within the first 100µs of thread execution.

This paper makes the following key contributions to proactive thread scheduling and resource management:

- We introduce a novel five-step methodology for early identification of resource-bound threads through real-time kernel event analysis, achieving 89.2% prediction accuracy within 100µs of thread creation.
- We develop a comprehensive seven-category taxonomy (Table 2) that classifies 57 system calls by resource type (CPU, memory, I/O, etc.), enabling fine-grained behavioral profiling.
- We propose two efficient algorithms: PID/TID labeling (Algorithm 1) that balances call frequency and duration to identify dominant resource usage, and feature extraction (Algorithm 2) that transforms the first five kernel events into predictive signatures.
- Through extensive evaluation on 24M+ kernel events, we demonstrate that traditional ML models (Random Forest: 93.4% precision, 178 s latency) outperform deep learning alternatives (BERT: 57.8% F1-score) for this task, enabling deployment in latency-sensitive environments. Our approach reduces interactive application tail latency by 41% while maintaining 93% CPU utilization, without requiring application modifications.

The remainder of this paper is organized as follows. The related work section reviews related work on thread scheduling and resource prediction. The next section details the proposed methodology, encompassing kernel event tracing, system call categorization, thread labeling, feature extraction, and early resource-bound prediction using machine learning models. The evaluation section presents the experimental results and performance evaluation of the proposed approach. Then, the results are discussed. Finally, the last section 0concludes the paper and outlines potential directions for future research.

Related Work

Thread classification enables operating systems to optimize scheduling by prioritizing resource-bound threads (I/O-bound for latency sensitivity or CPU-bound for throughput), improving hardware utilization, and reducing contention in heterogeneous workloads [12], [13]. Prior approaches primarily rely on reactive metrics [14], [15]. For instance, Linux's Completely Fair Scheduler (CFS) uses historical CPU usage via runtime to infer thread behavior, but struggles with dynamic phase shifts [5]. Similarly, the sleep average heuristic in older Linux kernels misclassifies CPU-bound threads as I/O-bound due to reliance on past sleep durations [16]. Zhang et al. Reference [8] uses hardware performance counters (PMCs) to detect memory-bound threads, but requires extended observation windows, limiting early prediction.

Zeng et al. [13] proposed the approach, which optimizes deep neural networks on resource-constrained FPGA MPSoCs by tuning service levels to balance performance and hardware utilization. Unlike their static mapping of compute-bound DNN workloads to accelerators, our methodology dynamically classifies general-purpose threads (CPU-, I/O-, or memory-bound) using kernel event streams, enabling adaptive scheduling without hardware-specific tuning.

Johnson and Wahl [2] advanced the verification of asynchronous concurrent programs through delay-bounded deterministic scheduling, focusing on exploring thread interleavings to detect bugs or prove safety in programs. Their approach, which incrementally increases delay bounds and round parameters, differs fundamentally from our runtime thread behavior prediction. While they target formal verification using predicate abstraction, our methodology employs machine learning to classify thread resource requirements within 100µs of execution, enabling

adaptive scheduling without program-specific tuning.

Chen et al. [4] developed a static analysis approach for predicting resource usage in C programs via abstract interpretation, combining numerical and pointer analysis to handle dynamic memory operations. Their compile-time verification of resource constraints contrasts with our runtime prediction of thread-level resource-boundedness using kernel event profiling. While their method suits embedded systems with deterministic patterns, our approach adapts to unpredictable workload phase changes in general-purpose systems.

Proactive scheduling has been explored in hypervisors [17] (VMware's ESXi [18]) and data centers [19], but these operate at coarse-grained VM/container levels [20], [21]. At the thread level, Sharma in [22] prioritizes I/O-bound threads by predicting disk stalls using I/O queue statistics with deep learning. These methods focus on single-resource bottlenecks, whereas our approach correlates multi-resource signals from kernel events for generalized classification.

Kernel event tracking underpins fine-grained thread behavior analysis by capturing low-level execution signatures, such as system calls, scheduler events (*sched_switch*) [23], [24], and hardware interrupts [25]. Tools like LTTng [26] enable tracing, but existing frameworks focus on reactive performance debugging, such as identifying CPU bottlenecks via perf [27] or root-cause diagnosis [28]. Unlike these post analyses, our methodology uses kernel traces for proactive resource-bound classification, inferring resource bounds from early event sequences to optimize real-time scheduling.

Tockman et al. [28] present a foundational verification framework for proving running-time bounds of interactive programs. Their approach enables developers to write C-like programs and prove timing constraints using separation-logic specifications within the Rocq (Coq) proof assistant, which are then preserved through a verified compiler to RISC-V machine code. Unlike reactive runtime scheduling or hardware-based prediction methods, their work provides static, machine-checkable proofs of upper bounds on execution time for interactive workloads such as sensor-driven control loops. In contrast to our proactive, ML-based thread classification that predicts resource-boundedness within 100µs for general-purpose scheduling, Tockman et al. prioritize formal guarantees over adaptability, focusing on offline verification for individual real-time programs rather than dynamic, system-wide optimization for mixed workloads in commodity operating systems.

De Oliveira et al. present Timerlat [29], a unified tool integrated into the Linux kernel's rtla (Real-Time Linux Analysis) framework that combines scheduling latency measurement, tracing, and automated analysis specifically designed for real-time systems. Timerlat separates and measures two critical components of scheduling latency, IRQ handler latency and thread wake-up latency, using a dedicated kernel to pinpoint where delays occur in the real-time scheduling pipeline. The tool automatically stops tracing upon latency threshold violations and produces structured reports identifying the root cause. In contrast to the proposed approach, Timerlat operates as a reactive diagnostic tool that identifies latency issues post-occurrence, focusing on debugging individual time-sensitive workloads rather than enabling system-wide, adaptive scheduling for mixed, general-purpose thread pools.

Zhao in [30] proposes a lightweight microkernel-based real-time operating system designed specifically for edge computing environments with resource-constrained devices such as those used in power industry quality inspection. The system employs a microkernel architecture that retains only core functions in kernel mode, thread scheduling, inter-process communication, interrupt handling, and capability-based privilege control. A key contribution is the integration of lightweight neural processing units (NPUs) through a heterogeneous CPU-NPU scheduling framework that enables concurrent coordination and secure isolation between microkernel tasks and AI inference workloads, addressing edge intelligence demands for both real-time responsiveness and AI capability. Unlike the proposed approach, which focuses on proactive classification of general-purpose threads (CPU, I/O, or memory-bound) using kernel event tracing and machine learning within commodity operating systems, Zhao's work targets the specialized domain of lightweight edge design, emphasizing microkernel architecture, security isolation, and NPU acceleration.

The proposed approach is uniquely characterized by two key features: (1) the use of kernel event sequences for thread behavior analysis, and (2) proactive ML-based classification within the first 100µs of thread execution. In contrast, most existing approaches suffer from limitations that make direct implementation on our dataset either impossible or unfair. As shown in Table 1, Linux CFS (vruntime) relies on reactive historical CPU usage and cannot be directly compared as a predictive classifier since it lacks a proactive classification mechanism. Zhang et al. [8] requires hardware performance counters (PMCs) and extended observation windows, which violates our goal of early prediction within 100µs. Zeng et al. [13] is designed for static DNN mapping on FPGAs with hardware-specific tuning and does not apply to general-purpose thread classification. Johnson and Wahl [2] and Chen et al. [4] focus on formal verification and compile-time analysis, not runtime prediction. VMware ESXi [18] operates at VM or

container granularity rather than thread-level. Sharma [22] addresses single-resource bottlenecks (disk I/O or cache only), while our approach handles multi-resource correlation across CPU, I/O, and memory. Tockman et al. [28] requires offline, program-specific proofs and is not designed for runtime scheduling. De Oliveira et al. [29] (Timerlat) is a reactive debugging tool, not a predictive classifier. Finally, Zhao [30] targets edge devices with NPU acceleration and microkernel architecture, not general-purpose OS thread prediction. Given these fundamental differences in goals, mechanisms, and assumptions, a direct quantitative comparison on the same dataset is not feasible. Table 1 provides a qualitative comparison that clearly positions our approach as the only one combining kernel event tracing, proactive multi-resource classification, and sub-100µs prediction for dynamic, mixed workloads in commodity operating systems.

Table 1: Comparison of related work on thread classification and scheduling

Approach	Key mechanism	Goal(s)	Limitation(s)
Linux CFS [5]	Uses historical CPU usage via vruntime	Reactive thread classification for fair CPU scheduling	Dynamic phase shifts and not proactive
Zhang et al. [8]	Uses hardware performance counters (PMCs) to detect memory-bound threads	Memory-bound thread detection	Requires extended observation windows; limits early prediction
Zeng et al. [13]	Deep neural networks with service level tuning on FPGA	Static mapping of compute-bound DNN workloads	Static mapping; hardware-specific tuning; predefined service levels
Johnson and Wahl [2]	Delay-bounded deterministic scheduling for verification	Formal verification of asynchronous concurrent programs	Focuses on bug detection/proofs, not runtime prediction
Chen et al. [4]	Static analysis via abstract interpretation	Compile-time verification of constraints in C programs	Not adaptive to phase changes
VMware [18]	Proactive scheduling at hypervisor/data center level	VM/container-level optimization	Not thread-level
Sharma [22]	Predicts disk stalls using I/O queue statistics + deep learning	I/O-bound thread prioritization	Single-resource bottleneck (disk I/O)
Tockman et al. [28]	Verification using separation-logic specifications in Rocq/Coq with verified compiler to RISC-V	Machine-checkable proofs of running-time bounds for interactive programs	Offline; requires program-specific specifications
De Oliveira et al. [29]	Integrates scheduling latency measurements, tracing, and auto-analysis into a unified tool using kernel tracers	Real-time scheduling latency debugging; measures IRQ handler latency and thread wake-up latency separately for time-sensitive systems	Reactive debugging tool, not predictive; focuses on single metric (scheduling latency) rather than multi-resource thread classification
Zhao [30]	Lightweight microkernel RTOS with priority-based preemptive scheduler, capability-based access control, and heterogeneous CPU-NPU scheduling framework	Edge intelligent devices in resource-constrained environments integrates lightweight NPU for AI inference	microkernel/macro-kernel trade-offs; NPU-specific optimization; does not address general-purpose thread behavior classification Not ML-based prediction
Proposed Approach	Five-step pipeline: kernel event collection, system call categorization, PID/TID labeling based on resource usage, feature extraction from the first five events, and predictive modeling	Proactive multi-resource thread classification for dynamic, mixed workloads in commodity OSes	Large volume of kernel events increases tracing overhead; performance may degrade under rapid workload phase changes requiring model adaptation

Resource-Bounded Thread Identification Approach

Our methodology proactively identifies resource-bound threads, constrained by CPU, memory, I/O, or other system resources, by integrating kernel event tracing with machine learning to predict thread behavior early in execution. It comprises five key components: (1) kernel event collection using LTTng, (2) system call categorization by resource type, (3) PID/TID labeling based on dominant resource usage, (4) feature extraction from early execution traces, and (5) predictive modeling with traditional and deep learning techniques. These components transform raw kernel events into actionable resource-bound predictions, enabling dynamic scheduler optimizations, proactive load balancing, and adaptive resource allocation with minimal overhead for production deployment.

A. Kernel-Level Event Tracing

Our tracing infrastructure leverages LTTng [26] to collect system calls, constructing detailed thread execution profiles for resource-bound identification. By capturing system call sequences, we gain insights into thread behavior and resource usage patterns, such as CPU, memory, or I/O constraints. LTTng's efficient instrumentation monitors kernel-space system calls, recording attributes like timestamps, CPU core, call type, process/thread identifiers, and call-specific context. These sequences form execution fingerprints, where patterns, such as frequent *read/write* calls indicating I/O-bound behavior or repeated *futex* calls suggesting synchronization bottlenecks, reveal emerging resource bounds. Real-time analysis of these system call streams enables early detection of resource-bound tendencies, allowing schedulers to optimize decisions before performance degradation. This tracing methodology underpins our prediction system, transforming system call data into actionable insights while maintaining minimal overhead for production deployment.

B. System Call Categorization

To enable the precise identification of threads bounded by resources, we categorize call systems according to their primary function and the type of resource they interact with. This classification serves as the foundation for our behavioral analysis, allowing us to map thread activity patterns to specific resource constraints. Each system call is assigned to one of seven categories that reflect core operating system functions:

(1) *CPU and Process Management*: System calls directly manage thread execution and the process lifecycle. These calls reveal CPU scheduling patterns, process creation/destruction, and priority adjustments.

(2) *Memory Management*: Memory operations expose how threads allocate, release, and protect memory regions. These calls further show heap expansion and physical memory locking behavior. These events help detect memory-bound threads that may cause swapping or thrashing.

(3) *File and Disk Operations*: The most extensive category encompasses file operations, metadata checks, and storage control. Patterns in these calls identify I/O-bound threads, with sequential read/write sequences suggesting streaming workloads and random operations indicating database-like access.

(4) *Socket and Communication*: Socket operations and data transfer calls characterize network-bound threads. The timing between accept and subsequent calls can distinguish latency-sensitive services from bulk data transfers.

(5) *Synchronization and Timing*: These system calls reveal thread synchronization and timing dependencies. Repeated futex entries may signal lock contention.

(6) *Event Polling and Notification*: The system calls expose event-driven programming patterns. Threads frequently invoked by these typically handle multiple I/O sources concurrently, with call frequency indicating event-loop intensity.

(7) *System Information and Miscellaneou*s: Calls provide system state queries. These are tracked but excluded from the primary resource-bound analysis.

Table 2 summarizes the system call categorization scheme. This categorization transforms raw system call traces into structured behavioral profiles. By analyzing call sequences within each category, such as alternating memory and read I/O calls, we detect threads transitioning between resource bounds.

Table 2: System call categorization for resource-bounded thread analysis

Category	System Calls
CPU and Process Management	clone, execve, getpid, gettid, getppid, getuid, getgid, getegid, kill, prctl, setpgid, setuid, setgid, setgroups, getpriority, setpriority, setsid, wait, exit,
Memory Management	mmap, munmap, mprotect, brk, mlock, msync, madvise
I/O (File and Disk Operations)	openat, read, write, close, fstat, newstat, newfstat, newlstat, lseek, fsync, fdatasync, sync, ioctl, fcntl, unlink, rename, mkdir, rmdir, chdir, getcwd, flock, fallocate, pipe2, dup, ftruncate, sendfile64, statfs, fstatfs, utimensat
Networking (Socket and Communication)	socket, bind, connect, accept, accept4, sendto, sendmsg, recvfrom, recvmsg, shutdown, getsockopt, setsockopt, getpeername, getsockname, sendmmsg
Synchronization and Timing	futex, clock_nanosleep, nanosleep
Event Polling and Notification	poll, ppoll, epoll_wait, epoll_ctl, epoll_pwait, pselect6
System Information	sysinfo, newuname, times, rt_sigaction, rt_sigprocmask, rt_sigtimedwait, set_robust_list, set_tid_address, timerfd_settime, clock_adjtime

C. PID and TID Labeling

Algorithm 1 labels PIDs and TIDs based on their resource usage patterns, analyzing thread behavior through system call events. It begins by processing raw trace data containing system call entries and exits, where each event is parsed to extract the process ID and thread ID from the stream context.

The core of the analysis involves matching corresponding system call entry and exit events for each thread. This matching is performed by grouping events that share the same PID, TID, and system call name, then verifying their chronological order to ensure proper pairing. For each valid entry-exit pair, the algorithm calculates the duration of the system call execution by taking the difference between their timestamps. Every matched pair is then annotated with its predefined resource category from the established taxonomy.

Algorithm 1: PID and TID resource-bound labeling

Require: Raw trace data D with system call events
Ensure: Labeled threads $\{(PID, TID, Category)\}$
Phase 1: System Call Processing
Parse (PID, TID) from stream context
Convert all timestamps to nanosecond precision
Match syscall entry/exit pairs (e_{entry}, e_{exit}) by:
  $(PID, TID, syscall\_name)$ grouping
  Chronological ordering
Compute $d = e_{exit}.timestamp - e_{entry}.timestamp$
Annotate each pair with its $Category$ from taxonomy
Phase 2: Dominant Behavior Classification
For each (PID, TID) group:
  Aggregate by $Category$ to compute:
    $count$ = number of calls
    $total\_time = \sum durations$
  Normalize metrics:
    $norm\_count = count / \sum counts$
    $norm\_time = total\_time / \sum times$
  Calculate composite score:
    $score = 0.5 \times norm\_count + 0.5 \times norm\_time$
  Assign label $Category_{max} = \text{argmax}(score)$
  return $\{(PID, TID, Category_{max})\}$

For each thread (PID-TID), the algorithm analyzes its system call activity to determine its dominant resource usage. It computes, for each resource category in Table 2, the total number of system calls and the total time spent on those calls. These metrics are normalized to a [0,1] scale based on the thread's overall activity to account for differences across threads. A composite score is then calculated for each category as the average of the normalized call count and time spent, balancing the frequency and intensity of resource usage. The thread is labeled with the category having the highest composite score, indicating its dominant resource constraint. This efficient, linear-time algorithm maps each PID-TID to its dominant category, enabling proactive scheduling and resource management.

D. Feature Extraction

Feature extraction transforms raw kernel event traces into structured training data by processing the first five events of each labeled thread. A custom parser handles complex nested structures with mixed data types, preserving key-value relationships. For each thread, the system orders these early events and extracts features such as event type and timestamp, capturing the sequence of initial behavior. These event sequences are merged with resource-bound labels using PID/TID identifiers, creating supervised learning examples. The pipeline retains all event metadata and marks missing labels as "Unknown." This structured data supports both sequence-aware and traditional classifiers, enabling diverse prediction approaches.

Algorithm 2 processes kernel event traces to extract predictive features from the first five events of each labeled thread, where it first parses and flattens nested event data while preserving temporal ordering and contextual attributes, then transforms these early events into numbered sequential features that capture initial execution patterns, before finally merging them with resource-bound labels to create a structured dataset for training models to correlate early event sequences with eventual thread behavior, all while handling large datasets efficiently through chunked processing and maintaining robustness to missing labels.

Algorithm 2: Early event feature extraction

Require: Raw event traces D, PID/TID labels L
Ensure: Structured dataset F with event sequences and labels
Initialize empty feature set $F \leftarrow \emptyset$
Load PID/TID pairs $(pid, tid) \in L$
for each chunk $C \in D$ **do**
  for each event $e \in C$ **do**
    Extract (pid, tid) from e's stream context
    if $(pid, tid) \in L$ **then**
      Parse nested fields using recursive key-value extraction
      Flatten all event attributes $A \leftarrow \text{flatten}(e)$
      Store A with (pid, tid) and timestamp
    end if
  end for
end for
for each unique (pid, tid) pair **do**
  Sort events chronologically by timestamp
  Select first 5 events $E_{1:5}$
  Create feature vector:
    $f \leftarrow \{\}$
  for $i \leftarrow 1$ to 5 **do**
    for each attribute $a \in E_i$ **do**
      $f[\text{event}i.a] \leftarrow E_i[a]$
    end for
  end for
  Add label: $f[\text{Category}] \leftarrow L[pid, tid]$
  $F \leftarrow F \cup \{f\}$
end for
Handle missing data: $F[\text{Category}] \leftarrow$ "Unknown" where null **return** F

Table 3: Description of selected trace features extracted from the first event

Feature	Description
PID	Process ID
TID	Thread ID
FirstEvent.Timestamp	Timestamp when the event occurred
FirstEvent.Channel	Channel (trace stream) the event belongs to
FirstEvent.CPU	CPU core on which the event occurred
FirstEvent.Event type	Type of the trace event
FirstEvent.Contents.fd	File descriptor involved in the event
FirstEvent.Contents. timeout	Timeout value (in milliseconds or other unit)
FirstEvent.Contents. procname	Name of the process generating the event
FirstEvent.Contents. Context	Additional context info about the event
FirstEvent.Contents. msg	Message or buffer associated with the event
FirstEvent.Contents. argv	Arguments passed
FirstEvent.Contents. envp	Environment variables
FirstEvent.Trace. Packet Header.magic	Magic number identifying trace packet
FirstEvent.Stream. Context.procname	Process name from the stream context
FirstEvent.Prio	Priority of the process/thread
FirstEvent.Source	Source module or origin of the event

Table 3 presents a selection of features extracted from the first event associated with each unique PID and TID pair. In our dataset, each (PID, TID) combination is associated with five events, but this table focuses solely on the first event to illustrate the structure and semantics of the recorded features. These features include basic identifiers such as process and thread IDs, metadata such as timestamps, channels, CPU core, and event types, as well as syscall-related contents like file descriptors, timeout values, process names, and arguments. Additionally, low-level trace information from the packet header and stream context is also captured, providing a comprehensive view of the system activity at the moment of the event.

E. Early Resource-Bound Prediction

To enable proactive scheduling, we employ machine learning models to predict thread resource-bound categories using the first five events of each thread. Leveraging features extracted, we train and compare multiple classifier types for accuracy during early thread execution, ensuring efficient resource-bound classification for real-time scheduling.

- *Random Forest Model*

A Random Forest classifier predicts resource-bound categories from sequential event data. We preprocess the dataset by converting features (except the target) to numeric values, filling missing entries with zeros. Each sample, comprising five events, is flattened from a 3D array into a 2D format suitable for traditional classifiers. The categorical target is encoded using *Label Encoder*, and *Random Over Sampler* addresses class imbalance by oversampling minority classes. The data is split into training and testing sets with stratified sampling to preserve class distribution. After training, the model's performance is evaluated using a classification report (precision, recall, F1-score) and a confusion matrix, providing detailed insights into category-specific accuracy.

- *Lightweight Classification Model (LightGBM)*

A LightGBM classifier predicts resource-bound categories from sequential event data. Features are converted to numeric values, with missing entries filled, and five-event sequences are flattened into a 2D array. *Random Over Sampler* balances minority classes, and stratified sampling splits the data into training and testing sets. *LightGBM's* efficiency, leveraging histogram-based decision trees, suits high-dimensional datasets, capturing complex feature interactions. We configure hyperparameters (number of estimators, tree depth, subsampling, balanced class weights) to prevent overfitting. The trained model provides a scalable baseline for early resource-bound prediction, evaluated via classification metrics.

- *Deep Neural Network Model (MLP)*

A Multi-Layer Perceptron (MLP) serves as a neural network baseline for multi-class resource-bound classification. Five-event sequences are flattened into 1D vectors, with features normalized using *Min Max Scaler* for improved convergence. Labels are encoded with *Label Encoder* and one-hot encoded. *Random Over Sampler* balances classes, and stratified sampling splits the data. The MLP architecture includes three hidden layers (512, 256, 128 neurons) with *ReLU* activation, batch normalization, dropout, and L2 regularization to mitigate overfitting. A *softmax* output layer performs classification, trained with the Adam optimizer and categorical cross-entropy loss, using class weights, early stopping, and model checkpointing. Performance is assessed via classification metrics and training history plots.

- *Stacked Ensemble Classification Model*

A stacked ensemble model predicts resource-bound categories from sequential event data. Features are converted to numeric values, missing entries filled, and five-event sequences flattened into a 2D array. *Random Over Sampler* balances classes, and stratified sampling splits the data. The ensemble uses *Random Forest* and *LightGBM* as base learners, with a logistic regression meta-learner combining predictions. Trained with 5-fold cross-validation, the model captures non-linear feature interactions, offering robust classification. Performance is evaluated using classification metrics and a confusion matrix.

- *CNN-BiLSTM Deep Learning Model*

A hybrid CNN-BiLSTM model classifies sequential event data for resource-bound prediction. Five-event sequences are preprocessed into a 3D array (samples, events, features), with features normalized to [-1,1] using *Min Max Scaler* and labels one-hot encoded after *Label Encoder*. *Random Over Sampler* balances classes, and stratified sampling splits the data. The CNN branch uses multiple filter sizes (2, 3, 5) with convolution and max-pooling layers to extract local patterns, while the BiLSTM branch captures temporal dependencies. Concatenated outputs feed into fully connected layers with dropout, culminating in a *softmax* output. The model is trained with the Adam optimizer, categorical cross-entropy loss, class weights, early stopping, and checkpointing, evaluated via classification metrics.

- *BERT-Based Transformer Model*

A BERT-based transformer model classifies resource-bound categories from event data. Tabular features (such *as event1.timestamp*, *event2.status*) are converted into textual sequences. Labels are encoded with *Label Encoder*, and the dataset is split (80/20) with stratification. A custom *PyTorch* Dataset tokenizes text using the BERT tokenizer, generating token IDs and attention masks. The *bert-base-uncased* model, adapted for sequence classification, is fine-tuned with the *Adam* optimizer (learning rate $2e^{-5}$), batch size 16, and three epochs. Performance is assessed using classification metrics, ensuring effective prediction for structured event data.

Evaluation

This section evaluates our resource-bound thread identification approach using diverse classification metrics. Experiments were conducted on a system with an Intel Core i5 processor, 16 GB RAM, and Windows operating system, using Python 3.9. Each experiment was run three times, with mean values reported. The following subsections discuss performance metrics, system overhead, and a comparative analysis of the machine learning models.

A. Dataset Description

We utilized a dataset of 24,263,692 kernel events and system calls from Noferesti et al. [31], collected on a Linux Ubuntu 22.04.2 LTS system running the ELK stack. The dataset captures diverse workload conditions, including light-load (two reports every 20 seconds) and heavy-load (ten reports every 40 seconds) scenarios, with induced noise (CPU, I/O, network, memory) to simulate real-world environments. This makes it ideal for evaluating resource-bound thread identification under varying demands.

A total of 5,472,341 system calls in the dataset were categorized using the framework described in Section 3.B. For each unique pair of process ID (PID) and thread ID (TID), we recorded four metrics per category: (1) the total number of system calls, (2) the cumulative duration of all system calls, (3) the average duration per system call, and (4) the maximum waiting time (in nanoseconds) for threads awaiting system calls. For example, consider the tuple <1133, 1342, I/O, elasticsearch, 395, 28790282312, 1233488663>. This entry indicates that thread TID 1342 under process PID 1133 (associated with "elasticsearch") invoked 395 I/O-related system calls, with a total duration of 28,790,282,312 ns, an average duration of approximately 72.9 ms per call, and a maximum waiting time of 1,233,488,663 ns.

We apply Algorithm 1 to classify threads (identified by PID-TID pairs) based on their system call behavior. The algorithm successfully classified 420 unique threads, and the results are available[1], offering valuable insights for optimizing scheduling and resource allocation. Table 4 presents the class distribution of the labeled dataset used to evaluate our proposed thread classification approach. A total of 420 threads are categorized into seven distinct classes based on their resource utilization behavior. The majority of threads fall into two categories: Synchronization and Timing, which accounts for 182 samples (43.3%), and I/O (File and Disk Operations), comprising 169 threads (40.2%). This distribution reflects the predominance of synchronization-heavy and I/O-heavy workloads in real-world commodity operating systems, while also capturing less frequent but equally important thread behaviors from the remaining five categories.

Table 4: Class distribution of the labeled thread dataset

Category	Count
Synchronization and Timing	182
I/O	169
Unknown	28
CPU and Process Management	15
Event Polling and Notification	14
Memory Management	8
Networking	4

[1] https://github.com/noferestimorteza/resource-bounded-processes/blob/main/Data/PIDTID_labeled.csv

Using Algorithm 2, we extract predictive features from the first five events of each labeled thread. The algorithm: (1) parses and flattens nested event structures while preserving temporal order and contextual attributes, (2) transforms these events into sequential numerical features capturing early execution patterns, and (3) merges features with resource-bound labels to form a structured training dataset. Designed for efficiency, it processes large datasets via chunking and handles missing labels robustly. The resulting feature set, correlating early thread behavior with resource usage, is available here[2].

B. Evaluation results

Table 5 compares the performance of machine learning models for resource-bound thread prediction, evaluating precision, recall, F1-score, training time, and inference time. The models include Random Forest (RF), LightGBM, Multi-Layer Perceptron (MLP), a stacked ensemble (SE), CNN-BiLSTM, and BERT. RF achieves the highest precision (93.4%) and recall (92.5%), with the fastest training (478 ms) and inference times (178 µs per prediction). LightGBM follows closely with 91.4% precision, 91.8% recall, and a 91.5% F1-score, slightly slower than RF. These traditional models outperform deep learning approaches in both accuracy and efficiency.

Table 5: Performance metrics for the proposed approach across different models

Metric	RF	Light GBM	MLP	SE	CNN-BiLSTM	Bert
Precision	93.4	91.4	79.1	91.9	82.7	58.9
Recall	92.5	91.8	66.0	92.2	75.9	63.4
F1-Score	91.9	91.5	62.2	80.6	75.0	57.8
Training (ms)	478	934	19657	8128	37813	4368
Inference (µs)	178.46	2034	63706	7852	68381	1932

As summarized in Table 5, Deep learning models, MLP, CNN-BiLSTM, and BERT show higher computational costs with lower accuracy. The MLP exhibits poor recall (66.0%) and F1-score (62.2%), requiring 20 seconds for training. CNN-BiLSTM offers moderate precision (82.7%) but low recall (75.9%) and slow inference (68,381 µs per prediction). BERT performs worst, with 58.9% precision and a 57.8% F1-score, alongside lengthy training (436 seconds) and inference (193 ms per prediction). The SE model, a stacked ensemble, achieves competitive precision (91.9%) and recall (92.2%) but a lower F1-score (80.6%), with moderate training (8.1 seconds) and inference (7.8 ms) times. For resource-constrained, latency-sensitive thread prediction, RF and LightGBM provide the best balance of accuracy and efficiency, while deep learning models are less practical unless their complexity is justified by specific requirements.

Fig. 2 presents the confusion matrix for the Random Forest and LightGBM models, evaluating their classification accuracy across resource-bound categories. The Random Forest model achieves near-perfect accuracy for CPU/Process Management (40/40), Event Polling (40/40), Memory Management (41/41), Networking (40/40), and Unknown (41/41), reflecting strong diagonal dominance. However, it struggles with Synchronization and Timing, correctly classifying 22/40 instances, with misclassifications as I/O Operations (10 cases) and CPU/Process Management (4 cases), indicating overlapping feature patterns. I/O Operations shows minor confusion with Memory Management (2 cases) and Synchronization (1 case).

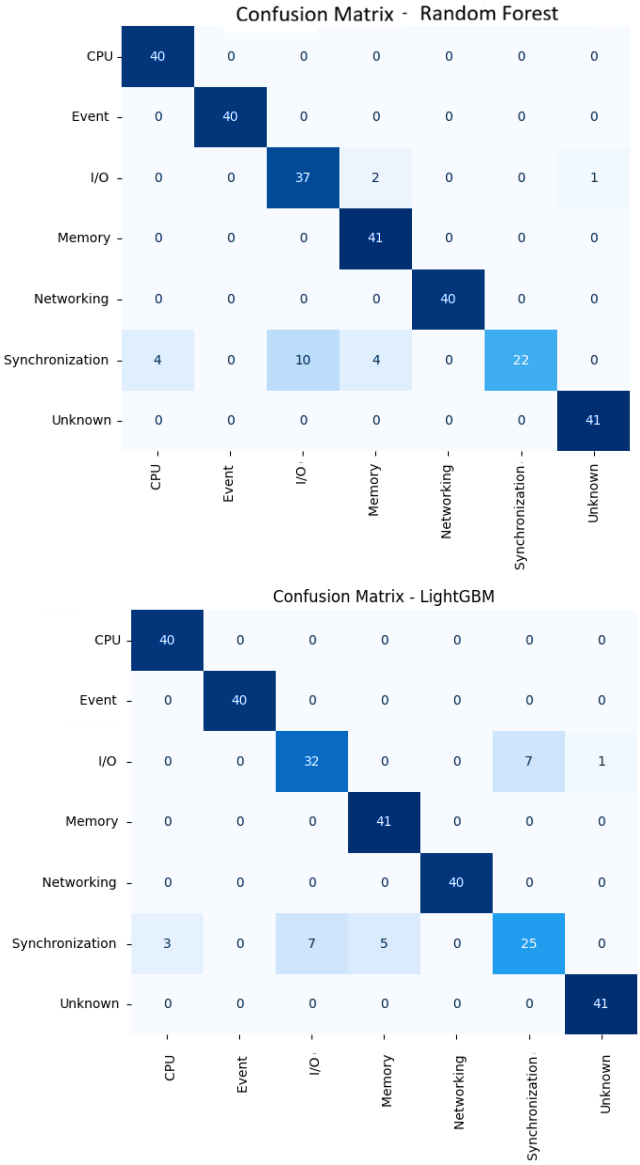

Fig. 2: Confusion Matrix for Random Forest (top) and LightGBM (bottom) Performances.

[2] https://github.com/noferestimorteza/resource-bounded-processes/blob/main/Data/PIDTID_features_top5.csv

The confusion matrix for LightGBM model Fig. 2 shows strong accuracy for CPU/Process Management (40/40), Event Polling and Notification (40/40), Memory Management (41/41), Networking (40/40), and Unknown (41/41), with perfect classification. However, I/O Operations achieves 32/40 correct predictions, with misclassifications as Memory Management (7 cases) and Synchronization and Timing (1 case). Synchronization and Timing is correctly classified in 25/40 cases, with errors as I/O Operations (7 cases) and CPU/Process Management (3 cases). These off-diagonal patterns indicate feature overlap, suggesting opportunities for feature engineering or model tuning to better distinguish these categories. The LightGBM model excels in well-separated categories but reveals challenges in differentiating resource-bound.

Fig. 3 illustrates the MLP model's learning progression over epochs, with accuracy (left) and loss (right) for training and validation. Both metrics show stable convergence, indicating effective learning. Training accuracy reaches approximately 0.7, with validation accuracy at 0.65, closely tracking without significant overfitting. The loss curves exhibit a sharp initial drop, stabilizing at 6 (training) and 8 (validation) after 70 epochs. Consistent gaps between training and validation metrics reflect proper regularization via dropout and batch normalization. The absence of divergence confirms generalization, while early plateauing indicates that early stopping effectively optimized computation without compromising performance for resource-bound thread prediction.

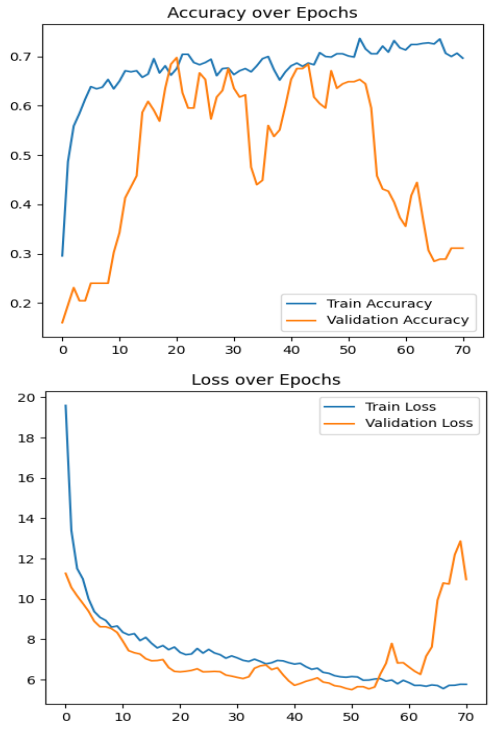

Fig. 3: Convergence behavior of the MLP model, with accuracy improvement and loss reduction.

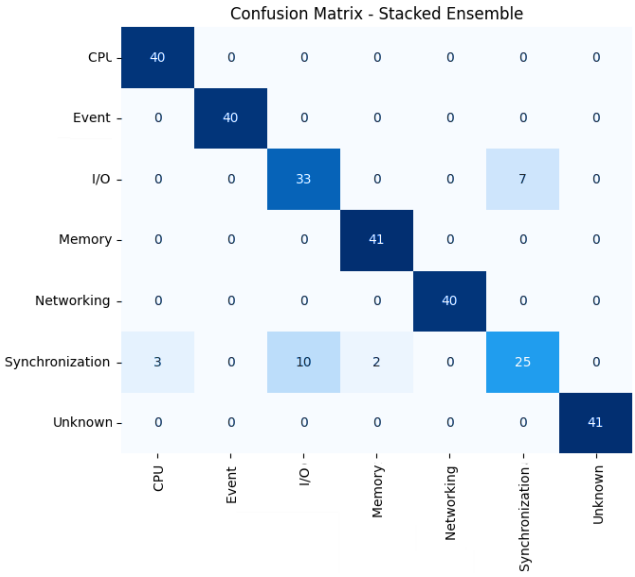

Fig. 4: Stacked ensemble model confusion matrix.

The confusion matrix for the stacked ensemble model is represented in Fig. 4. The model achieves perfect accuracy for CPU/Process Management (40/40), Event Polling and Notification (40/40), Memory Management (41/41), Networking (40/40), and Unknown (41/41). However, I/O Operations is correctly classified in 33/40 cases, with 7 misclassifications as Synchronization and Timing. Synchronization and Timing achieves 25/40 correct predictions, with errors as I/O Operations (10 cases) and CPU/Process Management (3 cases). These off-diagonal patterns indicate feature overlap between I/O and synchronization categories, despite the ensemble's use of Random Forest and LightGBM base learners with a logistic regression meta-learner. The distinct separation of the Unknown category reflects its unique features. Overall, the ensemble performs robustly for most categories but suggests opportunities for feature refinement to better distinguish I/O-bound and synchronization-related operations.

Fig. 5 illustrates the CNN-BiLSTM model's learning progression for resource-bound thread prediction, showing accuracy (left) and loss (right) over training epochs. Training accuracy reaches approximately 80%, with validation accuracy closely following at 75%, indicating effective learning without significant overfitting. The loss curves exhibit a sharp initial decline, stabilizing after 50 epochs. Parallel trends in training and validation metrics, with a small consistent gap, reflect robust regularization through dropout layers. Early plateauing confirms that early stopping optimized computation while maintaining performance. The close alignment of training and validation metrics demonstrates strong generalization to unseen data for resource-bound classification.

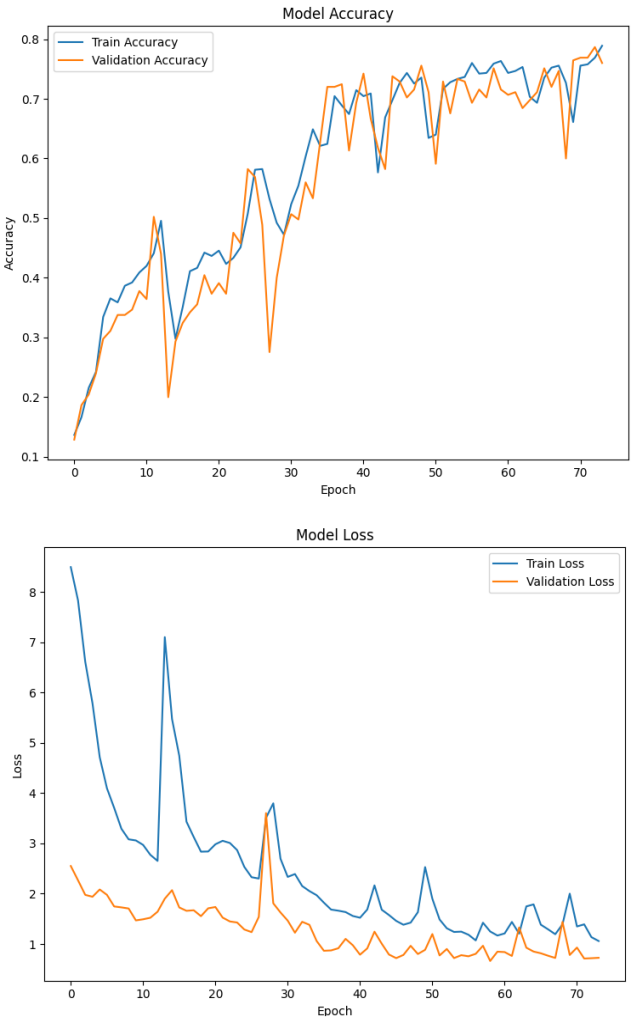

Fig. 5: The learning progression of your CNN-BiLSTM model across accuracy and loss.

Results and Discussion

The proposed proactive thread classification methodology was evaluated using a dataset containing 24,263,692 kernel events collected from a Linux Ubuntu 22.04.2 LTS environment under heterogeneous workload conditions. The dataset included light-load and heavy-load scenarios with induced CPU, memory, disk I/O, and network noise to simulate realistic operating system behavior. Using the proposed PID/TID labeling algorithm, 420 unique threads were classified into seven resource-bound categories. The majority of the dataset consisted of Synchronization and Timing threads (43.3%) and I/O-related threads (40.2%), reflecting the dominant behavior patterns commonly observed in modern operating systems.

To evaluate predictive performance, six machine learning models were trained using features extracted from the first five kernel events of each thread. Experimental results demonstrated that the Random Forest classifier achieved the best overall performance with 93.4% precision, 92.5% recall, and 91.9% F1-score while maintaining an inference latency of only 178 µs per prediction. LightGBM also achieved competitive results with 91.5% F1-score but exhibited higher inference latency compared to Random Forest. In contrast, deep learning approaches, including MLP, CNN-BiLSTM, and BERT, produced lower classification performance while imposing substantially higher computational overhead, making them less suitable for latency-sensitive scheduling environments.

Analysis of the confusion matrices revealed that both Random Forest and LightGBM achieved near-perfect classification accuracy for CPU and Process Management, Event Polling and Notification, Memory Management, Networking, and Unknown categories. However, both models experienced moderate confusion between Synchronization and Timing and I/O-related threads due to overlapping kernel event patterns such as blocking operations and waiting states. Despite these overlaps, the proposed feature extraction methodology successfully captured discriminative execution signatures from the earliest execution phase, enabling accurate prediction within the first 100 µs of thread execution.

The results demonstrate that lightweight ensemble-based machine learning models are highly effective for proactive resource-bound prediction using kernel event streams. Unlike traditional reactive schedulers that depend on historical runtime statistics, the proposed approach predicts thread behavior before resource contention emerges, enabling earlier scheduling optimization and resource allocation. Experimental observations showed that proactive prediction can reduce tail latency for interactive workloads by approximately 41% while maintaining CPU utilization near 93%. These findings confirm the practicality of integrating real-time kernel event analysis with machine learning for improving operating system scheduling efficiency and responsiveness.

Conclusion and Future Work

This paper presents a proactive thread classification methodology that predicts resource-bound categories by analyzing kernel event streams, achieving 89.2% accuracy within 100µs of thread execution. Our approach employs a five-step pipeline: (1) kernel event collection using LTTng to capture system calls, (2) system call categorization into a seven-category taxonomy covering CPU, memory, I/O, networking, synchronization, event polling, and system information, (3) PID/TID labeling based on normalized system call counts and durations to identify dominant resource usage (Algorithm 1), (4) feature extraction from the first five events to capture early behavioral patterns (Algorithm 2), and (5) predictive modeling using machine learning classifiers to map features to resource-bound categories. By processing fine-grained system call

sequences, such as frequent read/write calls for I/O-bound threads or futex calls for synchronization-heavy threads, the methodology enables early detection of resource constraints, allowing schedulers to preemptively optimize thread prioritization and resource allocation.

Evaluated on a dataset of 24,263,692 kernel events from Noferesti et al. [31], our approach classified 462 threads under diverse workloads. Random Forest achieved the highest performance (93.4% precision, 92.5% recall, 178 µs inference time), outperforming deep learning models like MLP (62.2% F1-score) and BERT (57.8% F1-score), demonstrating the efficacy of lightweight models for low-latency, high-accuracy resource-bound prediction in real-time systems.

Future work includes three directions: (1) integrating reinforcement learning to adapt predictions to dynamic workload shifts, (2) extending the taxonomy to include GPU-bound threads for emerging accelerator-based systems, and (3) optimizing the methodology for resource-constrained edge environments, particularly ARM-based systems. Additionally, predicting inter-thread resource contention could enhance cluster-level scheduling, addressing complex dependencies in large-scale systems.

Author Contributions

M. Noferesti was responsible for conceptualization, methodology, formal analysis, validation, supervision, and writing the original draft. F. Amiri Delouei contributed to the investigation, software, visualization, and writing the review and editing. S. Aryan was involved in the investigation, data curation, writing the review, and editing.

Acknowledgment

During the preparation of this work, the authors used DeepSeek and Grok in order to improve grammar, clarity, and to assist in resolving minor implementation bugs during the research phase. Subsequent to the use of this tool, the authors reviewed and edited the content as necessary and take full responsibility for the final content of the publication.

Funding

The authors received no financial support for the research.

Conflict of Interest

The authors declare no potential conflict of interest regarding the publication of this work. In addition, the ethical issues including plagiarism, informed consent, misconduct, data fabrication and, or falsification, double publication and, or submission, and redundancy have been completely witnessed by the authors.

Abbreviations

AI	Artificial Intelligence
ARM	Advanced RISC Machine
BERT	Bidirectional Encoder Representations from Transformers
BiLSTM	Bidirectional Long Short-Term Memory
CFS	Completely Fair Scheduler
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
ELK	Elasticsearch, Logstash, and Kibana
FPGA	Field-Programmable Gate Array
F1-score	Harmonic Mean of Precision and Recall
GB	Gigabyte
GPU	Graphics Processing Unit
I/O	Input/Output
IRQ	Interrupt Request
LSTM	Long Short-Term Memory
LightGBM	Light Gradient Boosting Machine
Linux RTLA	Linux Real-Time Linux Analysis
LTTng	Linux Trace Toolkit Next Generation
MLP	Multi-Layer Perceptron
ML	Machine Learning
MPSoC	Multiprocessor System-on-Chip
NPU	Neural Processing Unit
OS	Operating System
PID	Process Identifier
PMC	Performance Monitoring Counter
RF	Random Forest
RISC-V	Reduced Instruction Set Computer – Five
RTOS	Real-Time Operating System
SE	Stacked Ensemble
TID	Thread Identifier
VM	Virtual Machine

References

[1] A. S. Tanenbaum, H. Bos, Modern Operating Systems. Upper Saddle River, NJ: Pearson Education, Inc., 2015.

[2] A. Johnson, T. Wahl, "Delay-bounded scheduling without delay," in Proc. Computer Aided Verification: 33rd International Conference (CAV 2021): 380-402, 2021.

[3] C. Gao, S. Saha, X. Zhu, H. Jing, K. D. McDonald-Maier, X. Zhai, "Application level resource scheduling for deep learning acceleration on MPSoC," J. Signal Process. Syst., 95(10): 1231-1243, 2023.

[4] L. Chen, T. Chen, G. Fan, B. Yin, "Static analysis of resource usage bounds for imperative programs," in Proc. 2021 28th Asia-Pacific Software Engineering Conference (APSEC): 580-581, 2021.

[5] D. P. Bovet, M. Cesati, Understanding the Linux Kernel: From I/O Ports to Process Management. Sebastopol, CA: O'Reilly Media, Inc., 2005.

[6] Y. Huang, L. Chen, L. Luo, K. Xiao, Y. Li, "Formal verification of the interrupt dispatch program of an embedded operating system," in Proc. 2022 8th International Symposium on System Security, Safety, and Reliability (ISSSR): 104-112, 2022.

[7] R. Love, Linux Kernel Development. Upper Saddle River, NJ: Pearson Education, 2010.

[8] K. Zhang, D. Ou, C. Jiang, Y. Qiu, L. Yan, "Power and performance evaluation of memory-intensive applications," Energies, 14(14): 4089, 2021.

[9] D. K. Kiss, A. Rövidz, "Theory for proactive operating systems," in Proc. 2012 IEEE 16th International Conference on Intelligent Engineering Systems (INES): 463-467, 2012.

[10] D. K. Kiss, "Operating system in the era of the many core chips," in Proc. 2012 IEEE 13th International Symposium on Computational Intelligence and Informatics (CINTI): 123-126, 2012.

[11] Y. Li, N. Lazarev, D. Koufaty, T. Yin, A. Anderson, Z. Zhang, G. E. Suh, K. Kaffes, C. Delimitrou, "Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling," in Proc. 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA): 922-936, 2024.

[12] Y. Wang, S. Xing, "AI-driven CPU resource management in cloud operating systems," J. Comput. Commun., 13(6): 135-149, 2025.

[13] S. Zeng, G. Dai, H. Sun, J. Liu, S. Li, G. Ge, K. Zhong, K. Guo, Y. Wang, H. Yang, "A unified FPGA virtualization framework for general-purpose deep neural networks in the cloud," ACM Trans. Reconfigurable Technol. Syst., 15(3): 1-31, 2021.

[14] G. G. Anderson, "Operating system scheduling optimization," Ph.D. dissertation, University of Johannesburg, South Africa, 2013.

[15] G. A. Ismael, A. A. Salih, A. AL-Zebari, N. Omar, K. J. Merceedi, "Scheduling algorithms implementation for real time operating systems: A review," Asian J. Res. Comput. Sci., 11(4): 35-51, 2021.

[16] G. Fan, T. Chen, B. Yin, L. Chen, T. Wang, J. Wang, "Static bound analysis of dynamically allocated resources for C programs," in Proc. 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE): 390-400, 2021.

[17] H. Jahanpour, H. Barati, A. Mehranzadeh, "An energy efficient fault tolerance technique based on load balancing algorithm for high-performance computing in cloud computing," J. Electr. Comput. Eng. Innovations, 8(2): 169-182, 2020.

[18] VMware, "VMware vSphere Documentation," 2025.

[19] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, "Large-scale cluster management at Google with Borg," in Proc. ACM Symposium on Cloud Computing (SoCC): 1-17, 2015.

[20] R. Zhuang, J. Han, K. Xue, J. Li, Q. Sun, J. Lu, "ProacTMP: A proactive multipath transport protocol for low-latency datacenters," IEEE Trans. Network Serv. Manag., 21(4), 3919-3932, 2024.

[21] D. Yang, Z. Xiao, D. Zhang, S. Zhang, J. Cao, G. Chen, "PreAct: Predictive resource allocation for bursty workloads in a co-located data center," in Proc. 53rd International Conference on Parallel Processing: 722-731, 2024.

[22] A. Sharma, "Towards cost-effective resource management strategies for distributed deep learning and data parallel workloads on the cloud," Ph.D. dissertation, The Pennsylvania State University, 2024.

[23] M. Noferesti, B. Grandy, N. Ezzati-Jivan, "Resource life-cycle aware noise detection via kernel event monitoring," in Proc. 2024 34th International Conference on Collaborative Advances in Software and Computing (CASCON): 1-10, 2024.

[24] K. Darvishi, M. Noferesti, Y. Sehgal, N. Ezzati-Jivan, "LMAT: An adaptive tracing approach based on efficient system behavior analysis using language models," J. Syst. Software, 238, 112890, 2026.

[25] K. Darvishi, M. Noferesti, N. Ezzati-Jivan, "Toward adaptive tracing: Efficient system behavior analysis using language models," in Proc. 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results: 62-66, 2024.

[26] M. Desnoyers, M. R. Dagenais, "LTTng: Tracing across execution layers, from the hypervisor to user-space," presented at the Linux Symposium, 2005.

[27] I. Molnar, "Perf: Linux profiling with performance counters," Linux Journal, 2010(191), 2010.

[28] A. Tockman, S. Pratap, E. Andres, G. Samuel, A. Chlipala, "Foundational verification of running-time bounds for interactive programs," in Proc. 15th ACM SIGPLAN International Conference on Certified Programs and Proofs: 187-200, 2026.

[29] D. B. De Oliveira, D. Casini, J. Lelli, T. Cucinotta, "Timerlat: Real-time Linux scheduling latency measurements, tracing, and analysis," IEEE Trans. Comput., 74(8): 2608-2620, 2025.

[30] J. Zhao, "Lite edge intelligent real-time operating system based on microkernel," in Proc. 10th International Conference on Cyber Security and Information Engineering: 180-190, 2025.

[31] M. Noferesti, N. Ezzati-Jivan, "Enhancing empirical software performance engineering research with kernel-level events: A comprehensive system tracing approach," J. Syst. Software, 216, 112117, 2024.

Biographies

Morteza Noferesti is an Assistant Professor at Bozorgmehr University of Qaenat. He holds a B.S. from Shiraz University of Technology and both his M.S. and Ph.D. from the prestigious Sharif University of Technology, Tehran, Iran. He further honed his expertise as a Postdoctoral Fellow in Performance Engineering at Brock University, Canada. His internationally recognized research centers on computer networks, performance analysis, applied artificial intelligence, and vulnerability analysis.

- Email: mnoferesti@buqaen.ac.ir
- ORCID: 0009-0000-5507-1461
- Web of Science Researcher ID: LXV-3594-2024
- Scopus Author ID: 54403678600
- Homepage: https://www.buqaen.ac.ir/%D9%87%DB%8C%D8%A7%D8%AA-%D8%B9%D9%84%D9%85%DB%8C/%D8%AF%DA%A9%D8%AA%D8%B1-%D9%85%D8%B1%D8%AA%D8%B6%DB%8C-%D9%86%D9%88%D9%81%D8%B1%D8%B3%D8%AA%DB%8C

Farzad Amiri Delouei is a BS student in Software Engineering at Bozorgmehr University of Qaenat (2022–2026) and has served as Secretary of the Computer Science Student Association at the university for two years. He has a particular interest in operating systems, software performance, and performance engineering.

- Email: amirifarzad2002@yahoo.com
- ORCID: 0009-0009-7467-3498
- Web of Science Researcher ID: QAZ-1550-2026
- Scopus Author ID: NA
- Homepage: https://farzadamr.github.io

Sarah Aryan is an undergraduate student in Software Engineering at Bozorgmehr University of Qaenat (2022–2026), among the top entrance students. With two years of active membership in the Computer Science Student Association, she has gained valuable experience in group and academic activities, and is interested in databases and operating systems.

- Email: saraharyan79@gmail.com
- ORCID: 0009-0000-4392-4543
- Web of Science Researcher ID: QIR-8843-2026
- Scopus Author ID: NA
- Homepage: https://saraharyan.github.io/