IBIMA Publishing
Communications of International Proceedings
https://ibimapublishing.com/p-articles/45AI/2025/4533725/
Vol. 2025 (20), Article ID 4533725

# A Realistic Simulation Environment for Evaluating Dependency-Aware Scheduling in Multicore Systems*

Patryk SERAFIN

Faculty of Cybernetics, Military University of Technology,
00-908 Warsaw, Kaliskiego 2 Street, Poland
ORCID: 0009-0008-0573-1976

Correspondence should be addressed to: Patryk SERAFIN, patryk.serafin@wat.edu.pl

* Presented at the 45th IBIMA International Conference, 25-26 June 2025, Cordoba, Spain

## Abstract

Efficient task scheduling in multicore systems becomes increasingly complex when inter-task dependencies are introduced. Existing operating system schedulers prioritize fairness and responsiveness but lack mechanisms for evaluating dependency relationships during task dispatching. This leads to premature thread activation and resource inefficiencies, especially in dependency-rich workloads. While some solutions address this limitation through compiler-level or middleware-based strategies, they often assume full knowledge of the task structure in advance, making them impractical for dynamic or real-time environments. To address this gap, a configurable simulation environment has been developed to evaluate and compare task scheduling models in scenarios with evolving dependencies. The study focuses on a lightweight user-space execution strategy known as the Dependency-Aware Model (DAM), which uses a signal-based mechanism to defer task execution until all prerequisites have been met. The simulator supports configurable workload generation, real CPU-bound task execution, and thread-level scheduling compatible with both Windows and Linux systems. Execution metrics are collected in CSV format and visualized using matplotlib charts. Preliminary results demonstrate that the DAM approach significantly reduces idle processor occupation and improves total execution time in cases with dense dependency graphs. The environment offers a reproducible platform for testing advanced scheduling techniques and lays the foundation for future integration with adaptive AI/ML-based task management strategies.

**Keywords**: Multicore Systems, Task Scheduling, Dependency-aware Model, Simulation Environment

## Introduction

The evolution of multicore computing systems has increased the need for advanced methods of task scheduling and resource allocation. With the growing number of simultaneously executable threads, managing inter-task dependencies has become a critical factor influencing performance. While general-purpose operating systems such as Linux and Windows implement efficient schedulers focused on fairness, responsiveness and throughput, they typically lack mechanisms to consider logical task dependencies. As a result, threads may be dispatched despite being unable to proceed, leading to idle processor usage and increased contention.

Existing approaches to dependency handling are often embedded in compilers or middleware frameworks and rely on static analysis of task graphs. However, these techniques assume that the complete task structure is known a priori, which limits their applicability to dynamic and asynchronous workloads. In contrast, many real-world systems operate under conditions where tasks and their dependencies are generated at runtime. This makes traditional pre-scheduling strategies impractical in many modern applications. Numerous

**Cite this Article as:** Patryk SERAFIN, Vol. 2025 (20) "A Realistic Simulation Environment for Evaluating Dependency-Aware Scheduling in Multicore Systems " Communications of International Proceedings, Vol. 2025 (20), Article ID 4533725, https://doi.org/10.5171/2025.4533725

simulation tools have been proposed to study scheduling efficiency, including SimGrid, GridSim and RTSim. While effective for modeling distributed workflows or high-level scheduling policies, these tools often abstract away critical aspects of processor contention and thread behavior under real execution conditions. Furthermore, they do not provide mechanisms for evaluating signal-driven task activation or direct CPU occupancy using compute-bound workloads.

The simulation environment described in this work was developed to address these limitations by enabling the evaluation of dependency-aware scheduling strategies under realistic execution conditions. The core of the system is a lightweight user-space model that monitors dependency resolution using event-driven signaling. A task is dispatched only when all declared dependencies are satisfied, preventing premature activation and improving processor utilization. The conceptual foundations of this model have been discussed in Serafin (2025), while the present study focuses on the construction and evaluation of a simulation environment that supports configurable workload generation, real CPU-bound execution and comparative metric analysis.

The proposed simulator allows testing across various workload scenarios, with controlled variation of task set size, arrival timing, dependency probability, and CPU affinity. Unlike abstract models, the simulator performs actual computations, which better reflects real system behavior and enables measurement of execution time and contention. This also opens possibilities for future enhancements based on adaptive techniques. The integration of AI and ML mechanisms for learning optimal dispatching strategies based on historical patterns is identified as a prospective direction, especially given the increasing prevalence of hybrid intelligent scheduling frameworks (Shankar et al., 2013; Cheng et al., 2024). This work contributes a reproducible and extensible platform for evaluating dependency-aware scheduling policies in multicore systems, addressing the lack of tools that combine thread-level realism with flexible dependency modeling.

## Architectural Overview of the Simulation System

The simulation system is built as a modular, configurable environment intended for testing and comparing two distinct task scheduling approaches: traditional system-level scheduling and the Dependency-Aware Model (Serafin, 2025). Each component in the system performs a clearly defined role, contributing to a flexible and extensible simulation flow. The design supports reproducible experiments, dynamic task arrivals, and configurable execution environments across multicore systems.

To support scalability and clarity, the architecture separates simulation logic into independent but cooperating modules. These components communicate through defined interactions, facilitating maintainability and future extensions. An overview of the component-based architecture is presented in Fig. 1.

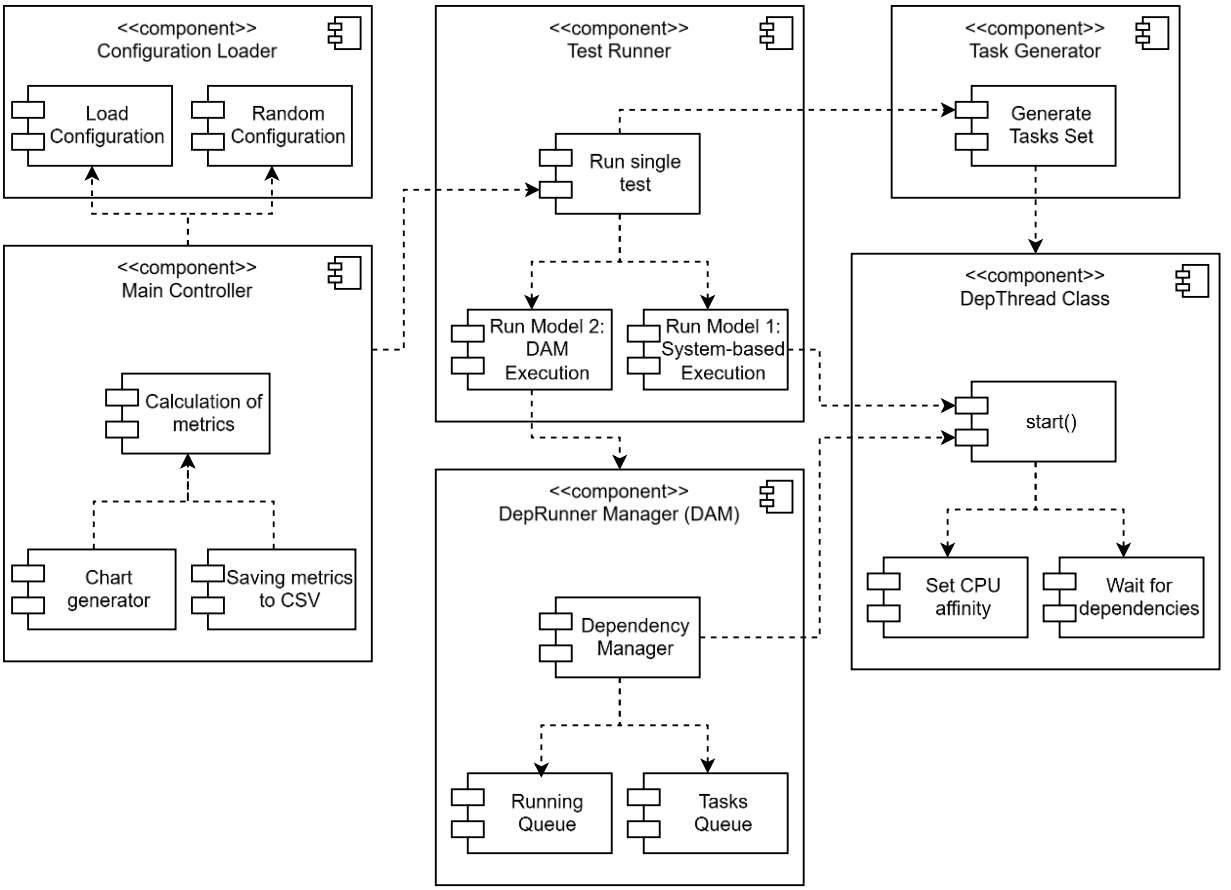


**Fig. 1 Component-based architecture of the simulation environment**

## *Component Roles and Interactions*

The simulation system comprises six core modules, each responsible for a specific stage of configuration, task handling, execution control, or result aggregation:

- Configuration Loader – Loads simulation settings either from a YAML configuration file or via randomized generation logic. The configuration includes all essential parameters for controlling simulation behavior, such as the number of tasks, timing settings, and probabilities for dependencies and CPU affinity.
- Main Controller – Serves as the primary orchestrator of the simulation. It manages the test execution flow, invokes the Test Runner, and processes the results. It also utilizes the matplotlib library for visualizing output metrics (The Matplotlib Development Team, 2025).
- Test Runner – Executes the simulation workflow for each test instance. It calls the ***Task Generator*** to create task sets for both standard execution and DAM-based scheduling models and sequentially executes them, collecting timing and dependency metrics.
- Task Generator – Constructs task sets based on the provided configuration. For each task, it assigns an execution time, probabilistically determines CPU affinity, and generates a dependency vector (based on termination signals from other tasks). Dependency structures are enforced to form valid directed acyclic graphs (DAGs), ensuring correctness.
- DepThread Class – A custom thread implementation derived from Python's threading.Thread class. It extends base thread functionality with native support for dependency signal checking and optional CPU affinity assignment. This class is used by all generated tasks across both execution models (Python Software Foundation, 2025).
- DepRunner Manager (DAM) – Core scheduling engine for the DAM approach. It maintains task queues, listens for dependency signal completions, and launches tasks only when all prerequisites are fulfilled. This component significantly reduces passive waiting and improves CPU efficiency.

These components are loosely coupled and interact through clear communication interfaces. Execution results and visualization generation are isolated within the ***Main Controller***, while scheduling logic remains encapsulated in the ***DepRunner***. To further illustrate inter-component behavior, Fig. 2 presents a sequence

diagram depicting the flow of control and data exchange across the architecture.

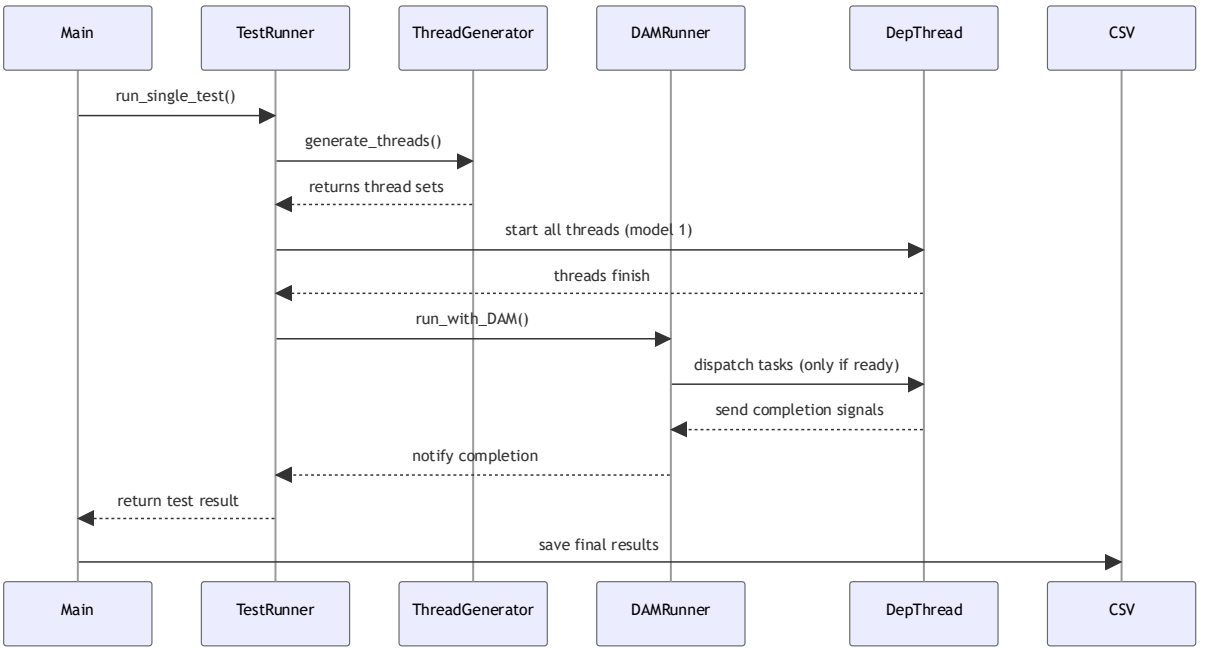


**Fig. 2 Sequence diagram of module interactions in the simulation environment**

## *Data Flow and Execution Lifecycle*

The simulation follows a structured lifecycle consisting of the following phases:

1. **Configuration Phase**
   Initiated by the *Main Controller*, this phase uses the *Configuration Loader* to parse a YAML file or randomly generate all required parameters. These include:
   - Number of tests and models,
   - Number of initial and additional tasks,
   - Dependency and CPU affinity probabilities,
   - Execution time and inter-arrival delay ranges.

2. **Task Generation**
   The *Test Runner* calls the Task Generator to build logically equivalent task sets for both scheduling models. Due to Python's threading model limitations, identical structures are created via independent instantiations rather than deep copies.
3. **Execution Strategy – Without DAM**
   In the first model, tasks are launched immediately upon arrival, regardless of dependency status. Each *DepThread* checks its dependency signals and waits passively if necessary. This may lead to inefficient resource usage under high dependency scenarios.
4. **Execution Strategy – With DAM**
   In the second model, tasks are passed to the *DepRunner*, which monitors their readiness. Tasks are dispatched only when all dependency signals are satisfied. The DAM runner listens for changes in task state and avoids unnecessary CPU usage through event-based triggering.
5. **Result Collection**
   After both execution strategies complete, the *Test Runner* sends timing results and execution metrics to the *Main Controller*, which processes, compares, and stores the results.

The task lifecycle is not only dependent on configuration parameters but also on the timing of incoming task sets and dynamically resolved dependencies. The DAM's event-driven loop ensures minimal overhead and avoids constant polling. Both standard and DAM-enhanced executions are visualized using timeline representations and bar charts. Section 0 provides examples of these visualizations, such as the chart in Fig. 4, which illustrates execution timelines, and Fig. 3, which presents bar chart output results generated using matplotlib.

## Configuration and Task Modeling

The simulation environment relies on a detailed and modular configuration system that allows for high flexibility in defining task sets and scheduling conditions. By supporting both manual and randomized configuration loading, it enables extensive testing of the Dependency-Aware Model (DAM) under diverse execution scenarios. This section presents the parameter structure and task generation logic, highlighting how correctness and realism are preserved through directed acyclic graphs (DAGs).

### *Parameter Structure and YAML Example*

Simulation parameters are stored in a structured YAML file that defines global settings, thread generation rules, and model comparison settings. These parameters control the number of tests to execute, task characteristics, scheduling constraints, and timing behavior. The configuration is handled by the Configuration Loader, which either parses the user-provided file or generates randomized parameters within defined bounds. Below is a representative configuration file (Listing. 1), illustrating key parameter groups:

```
1. general:
2.   tests_number: 1              # Number of tests in the simulation
3.   models_number: 2             # Number of models compared (standard + DAM)
4.   initial_threads_number: 20    # Number of tasks at simulation start
5.   additional_models_number: 3   # Number of tasks sets to arrive during runtime
6.   additional_threads_number: 15  # Number of tasks per additional set
7.   min_delta_time: 10            # Minimum delay (ms) between arriving tasks
8.   max_delta_time: 20            # Maximum delay (ms) between arriving tasks
9.
10. threads:
11.   min_execution_time: 10       # Minimum task execution time (ms)
12.   max_execution_time: 5000     # Maximum task execution time (ms)
13.   cpu_affinity_enabled: true   # Enables CPU affinity for tasks
14.   cpu_affinity_probability: 0.3  # Probability of assigning a CPU affinity
15.   dependencies_probability: 0.5  # Probability of generating dependencies
16.   max_dependencies: 20         # Maximum number of dependencies per task
```

**Listing. 1 Example of configuration file in YAML format**

Each parameter group influences a distinct part of the simulation pipeline:

- General Settings define the simulation's scale and temporal behavior.
- Thread Settings affect task complexity, interdependencies, and processor allocation.
- Probability Settings introduce variability and allow evaluation under stochastic conditions.

The ability to define precise test cases or explore randomized parameter spaces enables reproducibility and robustness in evaluating DAM-based scheduling.

### *Task Generation Logic and DAG Enforcement*

The Task Generator component is responsible for building the task sets used in simulations. It produces logically identical task structures for each model under evaluation (standard scheduler and DAM-enhanced), ensuring fair comparison. These sets include both:

- Initial Tasks – Introduced at the simulation start.
- Additional Task Sets – Arriving later at random intervals, as specified in the configuration.

Each task is constructed with the following attributes:

- Execution Time – Randomly sampled from a specified range to simulate real-world processing loads.
- Dependency Vector – Populated probabilistically with references to termination signals of preceding tasks.
- CPU Affinity (Optional) – Assigned based on probability, directing task execution to specific processor cores.

A key design feature is the ***dependency model***, which guarantees that all generated task sets form valid ***directed acyclic graphs (DAGs)***. This enforcement prevents circular dependencies and ensures that every task has a deterministic and reachable execution path.

Dependencies are generated as follows:

- For a task $T_i$ potential dependencies can only be drawn from tasks with lower indices ($T_j \; where \; j < i$), ensuring temporal consistency.
- A task can have up to ***max_dependencies*** dependencies, randomly selected without duplicates or self-references.
- Dependency signals are implemented via completion events, allowing tasks to listen for predecessor termination without requiring explicit task identifiers.

This structure supports scalable and accurate simulation of systems with heterogeneous and dynamic workloads, while preserving correctness through DAG validation.

# Simulation Runtime Behavior

The simulation environment is designed to replicate real-time task execution conditions in multicore systems by launching tasks that perform actual computations and consume processor time. Two distinct strategies are evaluated: a baseline model relying solely on the operating system scheduler and the enhanced Dependency-Aware Model (DAM). This section outlines how tasks are handled during simulation execution, focusing on the dependency resolution mechanism and temporal scheduling differences.

## *Event Loop and Dependency Checking*

In the DAM-enhanced model, task execution is coordinated by the DepRunner Manager (DAM), which maintains internal queues for both executable and waiting tasks. Rather than continuously polling the task list, the manager operates through an event-driven loop that reacts to system state changes in particular to the completion of tasks whose signals are monitored by dependent tasks.

Each task created for DAM is an instance of the custom DepThread Class, which extends Python's standard ***threading.Thread*** interface (Python Software Foundation, 2025). It includes built-in support for:

- Tracking a vector of dependency signals,
- Waiting on unfulfilled dependencies,
- Optional CPU affinity assignment before execution.

The DAM's event loop operates as follows:

- A global thread (DAM Runner) continuously listens for dependency signals.
- Upon receiving a signal (i.e., when a task completes), the runner inspects the queue of pending tasks.
- Tasks for which all dependencies have been resolved are moved to the ready queue and launched on available cores.

This strategy ensures that no task is launched prematurely. As a result, system resources are allocated only to tasks that are fully executable, improving CPU efficiency and eliminating unnecessary passive waits.

By contrast, in the baseline (non-DAM) model, all tasks are launched as soon as they are created. If a task has unresolved dependencies, it enters a passive waiting loop, consuming a thread and processor slot while waiting for signals from its predecessors. This behavior can lead to increased contention and reduced throughput, particularly under high-dependency workloads.

## *Task Execution Timeline*

To illustrate the behavioral difference between the two execution strategies, Fig. 3 presents a simplified Gantt chart showing the execution timeline of ten tasks under both approaches.

This visualization assumes a simplified scenario where each task has an equal execution time of one unit. The top section of the figure shows standard scheduling behavior, in which tasks are launched immediately regardless of dependency readiness. Critical tasks (those with unresolved dependencies) enter wait states, occupying CPU time while waiting for their required signals. The lower section illustrates DAM-based execution. Tasks are launched only when all dependencies are fulfilled. A background DAM runner manages the scheduling logic and transitions tasks into execution. As a result, no task occupies CPU time unnecessarily, leading to optimized processor utilization. It is important to note that Fig. 3 serves as a

conceptual illustration, not a direct output from the simulation system. It clarifies the fundamental distinction in execution semantics rather than showing empirical timing data.

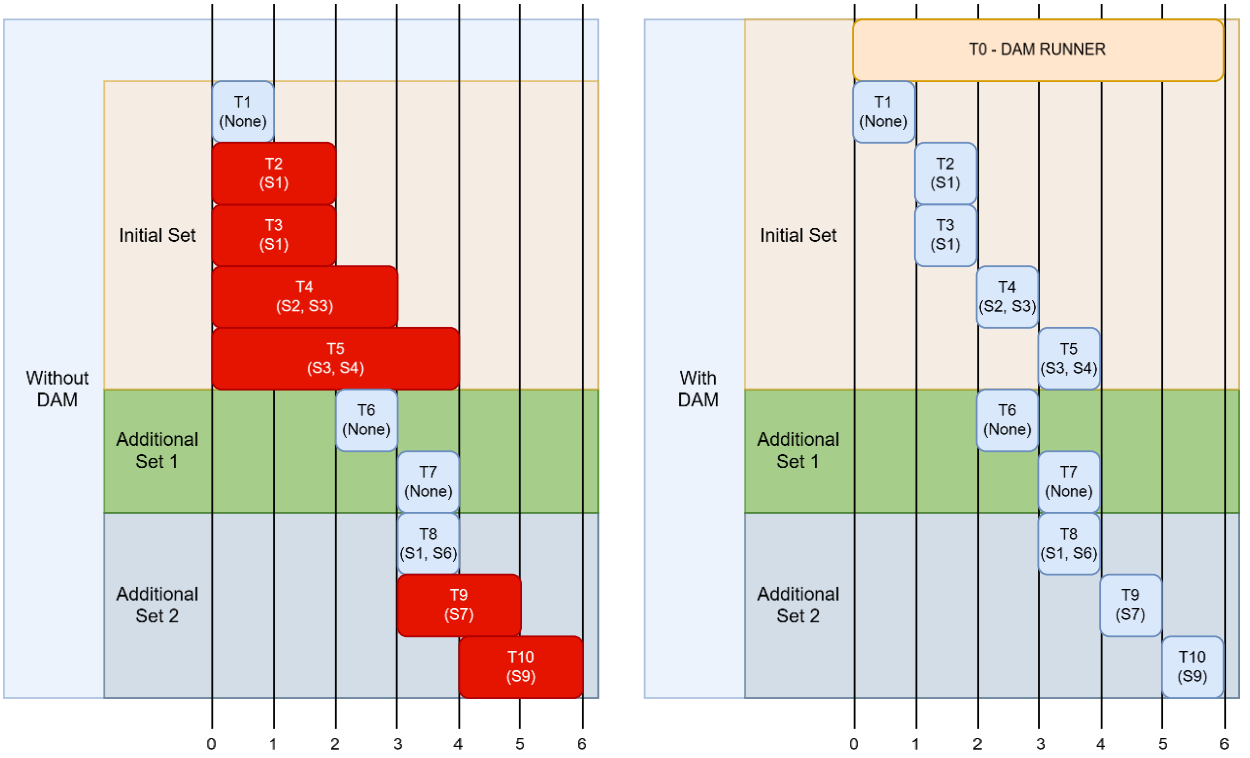


**Fig. 3 Task execution timeline with and without the Dependency-Aware Model (DAM)**

## *Output and Visualization*

The simulation environment records and summarizes execution metrics after the completion of each test series. This output facilitates a direct comparison between the standard execution model and the proposed Dependency-Aware Model (DAM). Data is stored in a structured CSV format and optionally visualized as summary charts to highlight performance differences across test cases.

## *CSV Metrics*

At the end of the simulation run after all defined tests have been performed, the results are aggregated and exported to a CSV file. Each row in the file corresponds to a full simulation instance with the following fields:

- timestamp – the time when the simulation was completed,
- avg_improvement – average percentage reduction in total execution time achieved by DAM compared to standard scheduling,
- avg_internal – average total execution time using standard scheduling (without DAM),
- avg_BFS – average total execution time using DAM,
- tests_num – number of tests run in the simulation,
- avg_dep – average number of dependencies per task set,
- avg_ex_time – average execution time of a single task,
- initial_thr – number of initial tasks present at simulation start,
- additional_tests – number of additional dynamic task sets,
- additional_thr – number of tasks in each additional set,
- total_thr – total number of tasks per test,
- min_delta_time / max_delta_time – range for delay between incoming additional tasks,
- min_ex_time / max_ex_time – range for task execution durations,
- aff_pro – CPU affinity assignment probability,
- dep_pro – dependency assignment probability,
- dep_max – maximum allowed dependencies per task.

These values enable reproducibility, performance profiling, and comparative analysis across simulation runs with varying configurations.

### *Matplotlib-Based Summary Charts*

To complement the CSV output, the simulation system leverages the ***matplotlib*** library to generate bar charts summarizing results from each simulation (The Matplotlib Development Team, 2025). The charts illustrate the percentage improvement observed for each individual test and annotate each bar with the exact improvement percentage and the total number of dependencies involved in the corresponding task set.
As shown in Fig. 4, each simulation test is represented as a single bar, with its height indicating the improvement achieved using DAM. Above each bar, two values are shown:

- The percentage gain in execution time,
- The total dependency count in square brackets.

This visualization helps assess the correlation between dependency complexity and scheduling gains from using the DAM model.

This figure demonstrates a sample simulation run configured with:

- 10 tests,
- 20 initial tasks,
- 3 additional sets of 15 tasks each,
- Task arrival delays sampled from 10–20 ms,
- Execution times between 10–5000 ms,
- CPU affinity assignment probability set to 0.3,
- Dependency probability set to 0.5,
- Maximum dependencies per task set to 20,
- Average execution time per task was 2402.88 ms.

Such visual feedback allows rapid interpretation of how model parameters influence overall performance and helps validate the effectiveness of dependency-aware scheduling under diverse workloads.

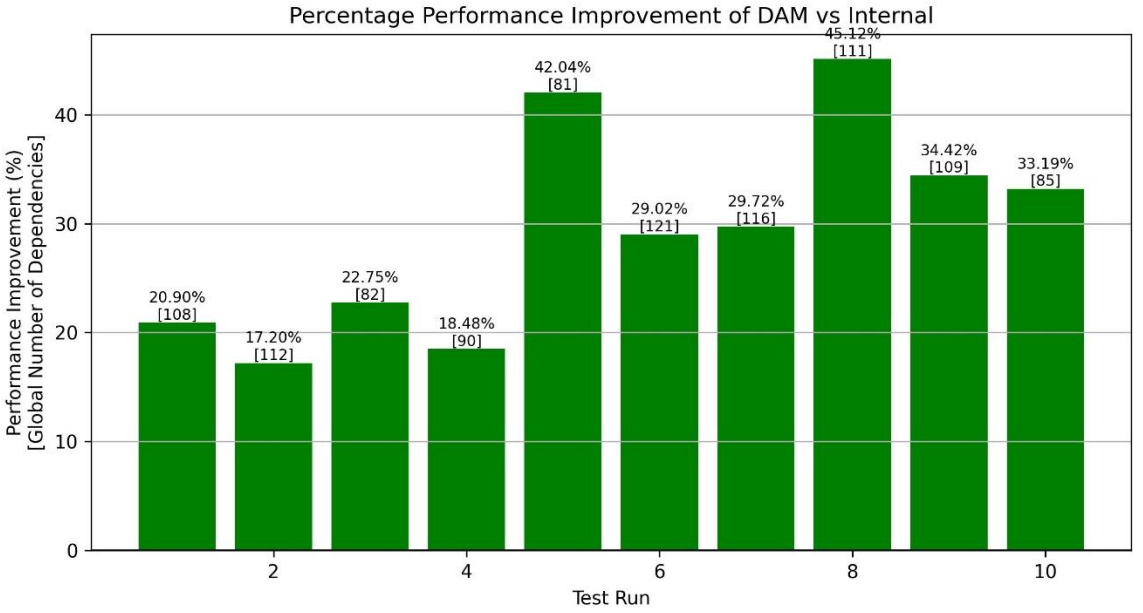


**Fig. 4 Example of simulation results chart showing time improvement per test case with task dependency statistics**

## Discussion

The developed simulation environment demonstrates a high level of adaptability and precision in evaluating

task scheduling strategies under controlled yet realistic conditions. Users can configure a wide array of parameters, including execution time distributions, dependency probabilities, and task arrival dynamics. This flexibility supports a broad spectrum of test cases and enables targeted exploration of various scheduling policies and dependency scenarios without requiring changes to the core simulation logic.

A major strength of the environment is its ability to execute tasks under real CPU load, which mimics the operational pressure found in production systems. Unlike abstract simulators that only estimate timing behavior, this system performs actual computation and creates genuine contention for system resources. As a result, it provides more accurate performance comparisons between execution strategies and highlights real-world inefficiencies in task scheduling during high-load conditions with complex dependency structures.
The component-based architecture ensures a clear separation of concerns. The Configuration Loader, Main Controller, and Test Runner form a coordinated pipeline, while DepRunner Manager and DepThread Class handle execution-level semantics. This design simplifies maintenance and makes it easier to extend the system in the future without disrupting its core workflow.

Although the environment supports the dependency-aware execution model (DAM), it remains agnostic to the internal logic of task workloads. Users can experiment with various computation patterns. By relying on wall-clock timing rather than artificial acceleration, the simulator avoids introducing measurement bias in execution durations and resource utilization.

Several limitations of the current implementation should be noted. The workload model only uses a single CPU-bound operation repeated for the duration of each task. This approach effectively generates processor stress, but it does not capture I/O-bound or hybrid tasks that may behave differently at runtime. In addition, the system always generates dependency vectors that form valid directed acyclic graphs. The simulation does not support cyclic dependency structures or advanced priority schemes currently.

Overall, the simulation environment provides a reliable foundation for evaluating and comparing task scheduling methodologies. Its modular design, high configurability, and use of real computing resources make it well suited for validating models like DAM and for broader research on parallel execution strategies in multicore systems.

## Conclusions and Future Work

A flexible and precise simulation environment was developed to evaluate task scheduling strategies in multicore systems. The simulator executes real CPU-bound workloads and compares a standard scheduler to a dependency-aware execution model (DAM). The Configuration Loader, Main Controller, Test Runner, DepRunner Manager, and DepThread Class each fulfil a distinct role and together enable full customization of experimental parameters including task arrival timing, execution duration distributions, dependency probabilities, and CPU affinity settings.

Initial results indicate that deferring task activation until all dependencies are satisfied reduces idle processor occupation and can decrease overall execution time in dependency-rich workloads. Visualizations such as Gantt charts and bar graphs of percentage improvement facilitate a clear understanding of scheduling behavior and performance differences between models.

To validate these findings and demonstrate the simulator’s effectiveness, tens of thousands of simulation trials will be conducted over diverse workload scenarios. These trials will vary initial thread counts, additional task sets, dependency densities, execution time ranges, and affinity probabilities. Statistical analysis of the aggregated results will confirm the efficacy of the DAM model and assess the reliability of the simulation framework. Plans also include extending the simulator to support more complex dependency structures such as cyclic or dynamically changing graphs and to integrate real operating-system scheduling interfaces.

## Acknowledgment

The work was financed by the Military University of Technology in Warsaw, Poland as part of the project No. UGB 531-000023-W500-22.

## References


- Microsoft. (2021), ‘About Processes and Threads’. [Online], Available at: https://learn.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads (Accessed: May 2025).
- Microsoft. (2022), ‘Process and Thread Functions’. [Online], Available at: https://learn.microsoft.com/en-us/windows/win32/procthread/process-and-thread-functions (Accessed: May 2025).
- Microsoft. (2021), ‘Scheduling’. [Online], Available at: https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling (Accessed: May 2025).
- Microsoft. (2025), ‘Windows Kernel-Mode Process and Thread Manager’. [Online], Available at: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-process-and-thread-manager (Accessed: May 2025).
- Python Software Foundation. (2025), ‘threading – Thread-based parallelism’. [Online], Available at: https://docs.python.org/3/library/threading.html (Accessed: May 2025).
- Serafin, P. (2025), ‘A Dependency-Aware Model for Task Execution Optimization in Multicore Systems’, in Proceedings of the 45th IBIMA Conference, Cordoba, Spain, June 2025.
- Shankar, S., Tamir, D. and Qasem, A. (2013), ‘Towards an operating system based framework for energy-efficient scheduling of parallel workloads’, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2013.
- The Linux Foundation. (2025), ‘Linux Scheduler – Kernel Documentation’. [Online], Available at: https://docs.kernel.org/scheduler/index.html (Accessed: May 2025).
- The Linux Foundation. (2010), ‘Workqueue – The Linux Kernel Documentation’. [Online], Available at: https://docs.kernel.org/core-api/workqueue.html (Accessed: May 2025).
- The Matplotlib Development Team. (2025), Matplotlib: Visualization with Python. [Online], Available at: https://matplotlib.org/ (Accessed: May 2025).