# Performance Evaluation of Multithreaded Systems

Brahmaiah Gandham, Salma Shaik, Rama Murthy Garimella and Praveen Alapati

October 21, 2024

# Performance Evaluation of Multithreaded Systems

Brahmaiah Gandham
*École Centrale School of Engineering*
*Mahindra University*
Hyderabad,India
brahmaiah.gandham@gmail.com

Salma shaik
*École Centrale School of Engineering*
*Mahindra University*
Hyderabad,India
salma20pcse004@mahindrauniversity.edu.in

Rama Murthy Garimella
*École Centrale School of Engineering*
*Mahindra University*
Hyderabad,India
rama.murthy@mahindrauniversity.edu.in

Praveen Alapati
*École Centrale School of Engineering*
*Mahindra University*
Hyderabad,India
praveena@cse.iitm.ac.in

***Abstract*—This paper presents a stochastic model-based evaluation of memory access performance in a multi-threaded system, where M threads independently attempt to access a shared memory segment. The model assumes that memory access times follow a Poisson process with exponentially distributed inter-access times. Key performance metrics, such as the probability of no contention, the probability of K threads contending, and the expected delay in accessing memory, are derived and analyzed. The results show that contention significantly increases with the number of threads, leading to sharp declines in performance. Numerical evaluations highlight the exponential increase in contention and memory access delays as the number of threads grows, emphasizing the importance of efficient synchronization mechanisms for scalable multi-threaded systems.**

***Index Terms*—performance evaluation, multi-threading, stochastic model, memory access, poison process.**

## I. Introduction

With the growing prevalence of multi-core and parallel processing architectures, efficient memory access has become a critical challenge in designing high-performance computing systems. In multi-threaded environments, threads often contend for shared memory resources, leading to delays and reduced system efficiency. As the number of threads increases, so does the likelihood of memory contention, which can significantly degrade overall performance. Understanding the stochastic behavior of thread contention and access delays is crucial for optimizing memory management and synchronization techniques [1], [2], [7], [9].

In multi-threaded systems, memory access times are often modeled using stochastic processes, particularly when dealing with concurrent threads attempting to access shared memory segments. A widely accepted approach is to model inter-memory access times using a Poisson process, where the access intervals follow an exponential distribution [3]. This approach captures the randomness and independent nature of memory access attempts, making it suitable for performance evaluation in parallel computing environments.

Past research has extensively studied contention in shared-memory systems, often focusing on lock-based synchronization mechanisms or non-blocking algorithms to reduce contention and improve efficiency [2], [4]–[6], [10]. However, while many studies focus on hardware-based solutions or software-level optimizations, the need for a generalized stochastic model to evaluate performance remains. This paper aims to fill this gap by presenting a stochastic model based on fundamental principles such as Occam's razor, offering a minimalistic yet effective approach to analyze memory contention in a multi-threaded environment.

The primary contribution of this paper is the development of a probabilistic framework to evaluate key performance metrics in multi-threaded systems, including the probability of no contention, the expected number of threads contending for memory access, and the expected delay in memory access. Using this model, we provide insights into how system performance degrades as the number of threads increases, emphasizing the exponential growth in contention and delays. Through numerical evaluations, we demonstrate the practical implications of these results and highlight the importance of optimizing thread synchronization mechanisms to mitigate performance degradation in highly concurrent systems.

This work builds on prior models and expands the understanding of contention in multi-threaded environments by offering a stochastic framework that can be adapted to various memory management scenarios. The results are particularly relevant for applications in cloud computing, real-time systems, and parallel scientific computing, where efficient memory access is critical for performance.

This research paper is structured as follows: Section II presents an analysis of Stochastic Model-based Performance Evaluation, followed by Section III, which focuses on Stochastic Model-based Numerical Analysis. Finally, Section IV provides conclusions regarding the stochastic model.

## II. Stochastic Model-based Performance Evaluation

In our stochastic model, we consider $M$ threads attempting to access a memory segment. The following modeling assumptions are made based on the principle of Occam's razor:

1) From the point of view of memory access, the threads are statistically independent.
2) The inter-memory segment access times are identically distributed for all threads. The common probability distribution is exponential with rate $\lambda$.
3) The time horizon for memory access is considered infinite. Effectively, based on assumptions (1) and (2), the memory access time points constitute a Poisson point process.
4) On accessing a memory segment, a thread locks the access for a certain minimum time $T_{\text{min}}$ and a maximum time $T_{\text{max}}$.

Based on the above modeling assumptions, we first consider the simple case:

The thread memory access times occur at the beginning of the time slot of length $T_{\text{min}}$ or $T_{\text{max}}$. A common clock synchronizes the $M$ threads, and hence effectively the time axis is slotted into slots of length $T_{\text{max}}$ or $T_{\text{min}}$.

## A. Performance Measures

We now evaluate the following performance measures based on the above modeling assumptions:

1) **Probability that during a slot no contention occurs:**

$$P(\text{no contention}) = \frac{e^{-M\lambda T_{\text{min}}}\left(M\lambda T_{\text{min}}\right)^1}{1!}$$

$$\theta = M\lambda T_{\text{min}} e^{-M\lambda T_{\text{min}}} \quad (\text{or} \quad M\lambda T_{\text{max}} e^{-M\lambda T_{\text{max}}})$$

2) **Delay probability distribution for number of slots required for successfully accessing a memory slot (random variable $Z$):**
Based on well-known facts related to the Poisson process, the discrete probability mass function is given by:

$$p\{z = m\} = (1-\theta)^{m-1}\theta$$

Hence, the discrete random variable $Z$ has a geometric probability mass function with success probability $\theta$.

3) **Probability that $K$ threads contend for memory access during the current slot:**

$$= \frac{e^{-M\lambda T_{\text{min}}}\left(M\lambda T_{\text{min}}\right)^K}{K!} \quad \text{for} \quad K \geq 1$$

4) **Probability that the delay in memory access exceeds $L$ slots:**

$$P(\text{delay} > L) = \sum_{j=L}^{\infty} p(z = j)$$

5) **By chebyshev inequality:**

$$P(\text{delay} > L) \leq \frac{ExpectedDelay}{L}$$

## B. Synchronization Effects

We realize that synchronization of threads requires additional effort. Without synchronization, the probability that there is no contention drops to a lower value. We now provide the detailed calculation.

## C. Inter-Memory Access Time

The inter-memory access time is assumed to follow an exponential distribution. The probability density function is:

$$f_x(y) = \lambda e^{-\lambda y}$$

The expected value of $Z$ is:

$$E(z) = \frac{1}{\lambda}$$

For example, if the inter-memory access time is 10 ms, then in one second, the number of memory requests $\lambda$ is:

$$\lambda = \frac{1000 \text{ ms}}{10 \text{ ms}} = 100$$

## D. Parameters for performance evaluation

- $\Lambda$ or equivalently $\frac{1}{\lambda}$
- $T_{\text{min}}$: Minimum time for which the memory segment is locked.
- $T_{\text{max}}$: Maximum time for which the memory segment is locked.
- Vary $K$, i.e., number of contending threads.

Let's now vary the number of threads $M$ from 1, 2, 4, 8, 16, 32, and 64 and calculate the corresponding values for the three performance metrics (i.e they are in geometric progression):

We will use the following assumptions:

$\lambda = 100$ (accesses per second) each thread accesses the memory segment once every 10 ms. $T_{min} = 0.01$ seconds (10 ms).

## III. Stochastic Model-based Numerical Analysis

TABLE I
PROBABILITY OF NO CONTENTION DURING A TIME SLOT FOR DIFFERENT VALUES OF $M$.

| $M$ (threads) | $P_{\text{no\_contention}}$ |
|---|---|
| 1 | 0.36788 |
| 2 | 0.27067 |
| 4 | 0.07326 |
| 8 | 0.00269 |
| 16 | $1.75 \times 10^{-6}$ |
| 32 | $4.88 \times 10^{-13}$ |
| 64 | $1.42 \times 10^{-27}$ |

Table 1 shows how the probability of no contention during a time slot varies as the number of threads ($M$) increases. With only 1 thread, the probability of no contention is around 0.36788, meaning there is no competition since only one thread is accessing memory. As the number of threads increases to 2, this probability drops to 0.27067, reflecting a greater likelihood of contention as more threads attempt to access memory simultaneously. For 4 threads, the probability decreases drastically to 0.07326, showing that contention becomes increasingly likely as the thread count rises. By the time $M = 8$, the probability of no contention is almost zero (0.00269), and with 16 or more threads, it effectively approaches zero. This trend highlights that with a larger number of threads, contention is almost inevitable.

TABLE II
PROBABILITY OF $K$ THREADS CONTENDING FOR MEMORY ACCESS DURING A TIME SLOT.

| $M$ (threads) | $K = 1$ Probability | $K = 2$ Probability |
|---|---|---|
| 1 | 0.36788 | 0.18394 |
| 2 | 0.27067 | 0.27067 |
| 4 | 0.07326 | 0.14652 |
| 8 | 0.00269 | 0.01075 |
| 16 | $1.75 \times 10^{-6}$ | $1.40 \times 10^{-5}$ |
| 32 | $4.88 \times 10^{-13}$ | $7.81 \times 10^{-12}$ |
| 64 | $1.42 \times 10^{-27}$ | $4.54 \times 10^{-26}$ |

Table 2 evaluates the probability of exactly 1 or 2 threads contending for memory access during a time slot, based on varying thread counts. When $M = 1$, only one thread is present, so the probability of 1 thread contending is 0.36788. For $M = 2$, the probability of both 1 thread and 2 threads contending is equal (0.27067 for each), reflecting the balanced likelihood of one or two threads attempting access. As the number of threads increases, the probability of only 1 thread contending decreases sharply; for $M = 4$, it is 0.07326, and for $M = 8$, it is only 0.00269. In contrast, the probability of 2 threads contending grows as more threads are added, though beyond a certain point (e.g., $M = 16$ or more), contention between multiple threads becomes almost certain, and the probability of only 1 or 2 threads contending becomes negligible.

TABLE III
EXPECTED DELAY IN SLOTS FOR SUCCESSFULLY ACCESSING MEMORY.

| $M$ (threads) | Expected Delay (slots) |
|---|---|
| 1 | 2.718 |
| 2 | 3.694 |
| 4 | 13.65 |
| 8 | 371.74 |
| 16 | 571428.57 |
| 32 | $2.05 \times 10^{12}$ |
| 64 | $7.04 \times 10^{26}$ |

Table 3 shows the expected number of slots required to successfully access memory (i.e., the delay), depending on the number of threads contending. With just 1 thread, the expected delay is 2.718 slots, meaning that memory access is relatively quick. As the number of threads increases, the delay grows exponentially. For 2 threads, the delay is 3.694 slots, while for 4 threads, it jumps to 13.65 slots, indicating a sharp rise in contention. By the time $M = 8$, the expected delay becomes very large, at 371.74 slots. For 16 or more threads, the delay becomes impractically long—571,428 slots for $M = 16$, and it reaches astronomical values for $M = 32$ and $M = 64$. This exponential increase in delay shows how quickly system performance deteriorates when many threads contend for memory access, making it clear that higher thread counts lead to severe delays without proper contention management.

The table reveals how the probability that the delay in memory access exceeds a given number of slots changes as the number of threads increases. For a single thread ($M = 1$), there is no contention, resulting in a probability of 1.0 for exceeding 1 slot, as the thread always has direct access to memory. As the delay threshold $L$ increases, the probability drops quickly, becoming negligible by $L = 32$.

TABLE IV
PROBABILITY THAT THE DELAY IN MEMORY ACCESS EXCEEDS $L$ SLOTS FOR UP TO 8 THREADS.

| $L$ | Threads=1 | Threads=2 | Threads=4 | Threads=8 |
|---|---|---|---|---|
| 1 | 1.000000 | 1.000000 | 0.999465 | 0.233594 |
| 2 | 0.632121 | 0.729329 | 0.926202 | 0.230911 |
| 4 | 0.252581 | 0.387946 | 0.795386 | 0.225565 |
| 8 | 0.040327 | 0.109766 | 0.586544 | 0.214959 |
| 16 | 0.001028 | 0.008787 | 0.318876 | 0.194086 |
| 32 | $6.68 \times 10^{-7}$ | 0.000056 | 0.094014 | 0.153663 |

With two threads ($M = 2$), contention slightly increases, but the system still handles delays well. The probability of delays exceeding $L = 1$ slot remains 1.0, though it gradually declines for higher values of $L$. By $L = 16$, the probability decreases to $0.008787$, indicating minimal delay, even with two contending threads.

At four threads ($M = 4$), contention becomes more apparent. The probability of delays exceeding $L = 1$ slot is $0.999465$, and by $L = 8$, it is more likely that delays exceed this threshold than not, with a probability of $0.586544$. Significant delays become more likely with moderate contention, as the system struggles to manage more threads.

With eight threads ($M = 8$), contention is more significant. The probability of delays exceeding $L = 1$ slot drops to $0.233594$, meaning that about 23% of the time, a delay of more than 1 slot occurs. For $L = 8$, the probability is still high at $0.214959$, reflecting frequent delays. Even for $L = 32$, the probability remains at $0.153663$, showing that delays of this magnitude are not uncommon with high contention.

Overall, as the number of threads increases, the likelihood of experiencing delays beyond a given number of slots rises sharply due to increased contention for memory resources. This highlights the importance of efficient synchronization and resource management in multi-threaded environments to prevent significant performance degradation.

Figures 1 and 2 show the relationship between interval time and execution time in a multi-threaded environment. By varying the interval time between successive memory requests while maintaining a fixed number of threads and memory requests, we aim to quantify the impact of this parameter on overall performance.

Our findings indicate a strong linear correlation between interval time and execution time. As the interval time increases, the total execution time also increases proportionally. This suggests that the time spent waiting for memory requests significantly contributes to the overall execution cost.

The observed relationship can be attributed to several factors, including thread scheduling overhead, memory access latency, and synchronization costs. When threads are forced to wait for longer intervals, they are more likely to be preempted by other ready threads, leading to increased context switching.

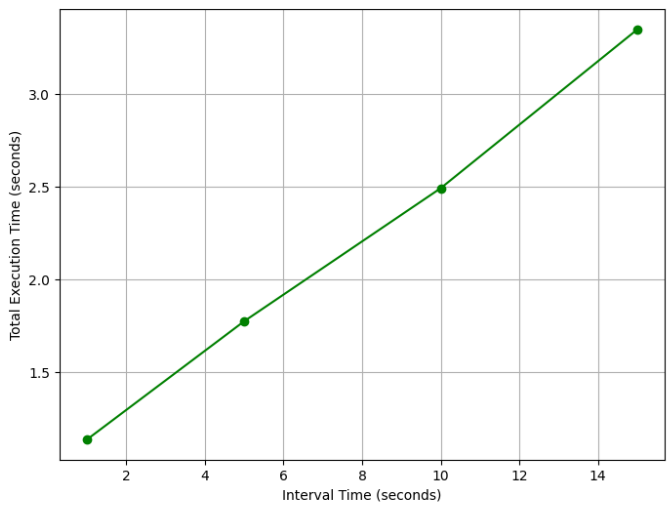

Fig. 1. Execution time by varying the interval time (number of memory requests is 10)

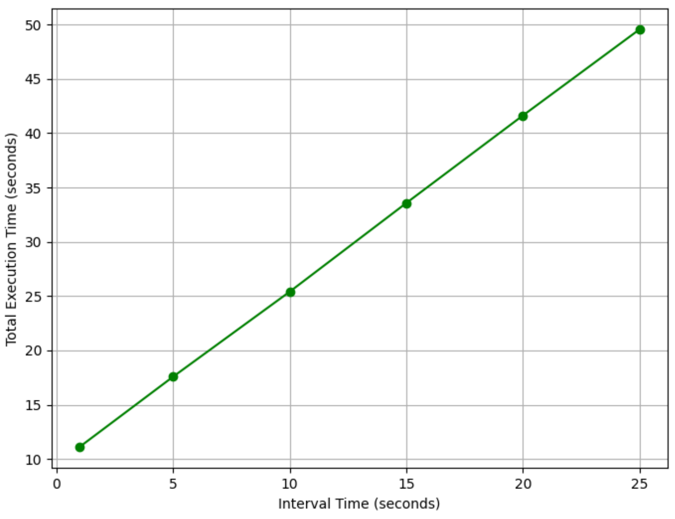

Fig. 2. Execution time by varying the interval time (number of memory requests is 100)

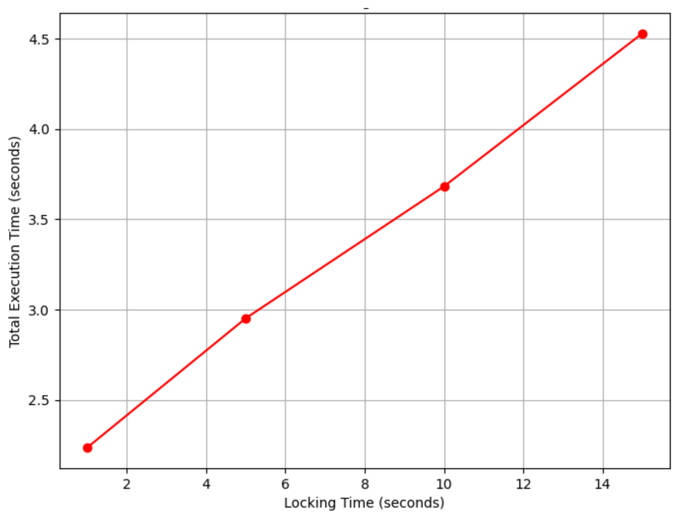

Fig. 3. Execution time by varying the locking time (number of memory requests is 10)

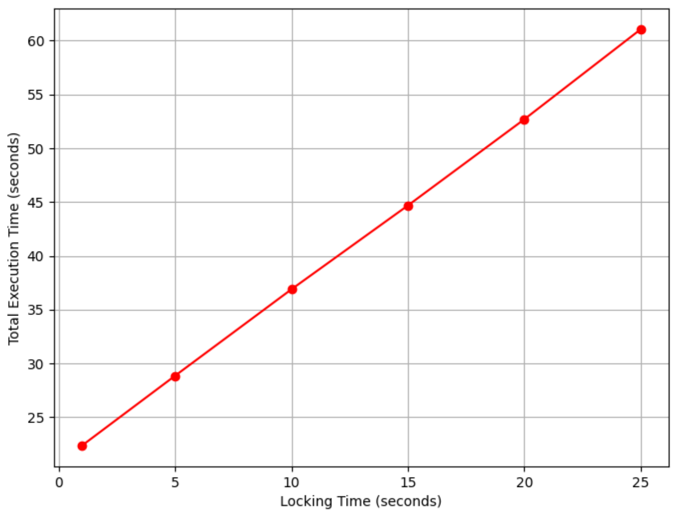

Fig. 4. Execution time by varying the locking time (number of memory requests is 100)

Additionally, longer intervals can increase the likelihood of cache misses and memory bus contention, further contributing to performance degradation.

These results underscore the importance of carefully considering interval time when designing and optimizing multi-threaded applications. By selecting appropriate interval times and employing techniques such as thread pooling and asynchronous programming, developers can effectively mitigate the negative impact of memory access latency and improve overall performance.

Figures 3 and 4 show the relationship between locking time and execution time in a multi-threaded environment. By varying the locking time while maintaining a fixed number of threads and memory requests, we aim to quantify the impact of this parameter on overall performance.

Our findings indicate a strong linear correlation between locking time and execution time. As locking time increases, the total execution time also increases proportionally. This suggests that the time spent on locking operations significantly contributes to the overall execution cost.

The observed relationship can be attributed to increased contention among threads competing for shared resources and the overhead associated with locking mechanisms. When threads are forced to wait for longer periods to acquire locks, they are less likely to make productive use of their allocated CPU time, leading to decreased system throughput.

These results underscore the importance of carefully considering locking strategies when designing and optimizing multi-threaded applications. By minimizing locking operations, using appropriate locking mechanisms, and identifying and addressing performance bottlenecks, developers can significantly improve the responsiveness and efficiency of their applications.

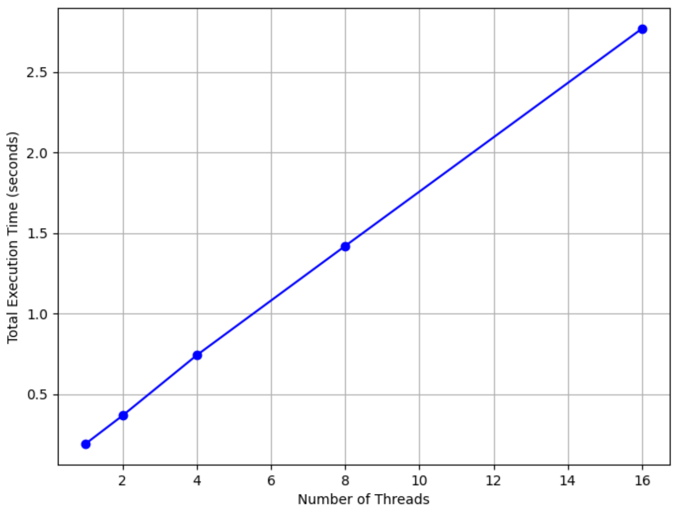

Fig. 5. Execution time by varying the threads (number of memory requests is 10)

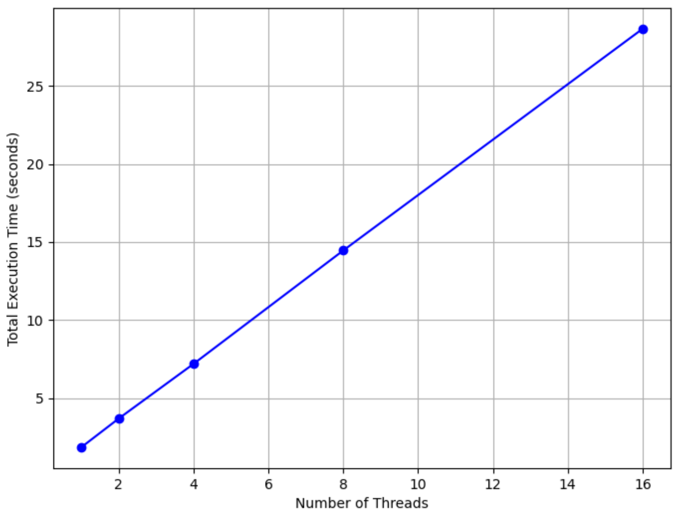

Fig. 6. Execution time by varying the threads (number of memory requests is 100)

Figures 5 and 6 the relationship between the number of threads and execution time in a multi-threaded environment. By varying the number of threads while maintaining a fixed number of memory requests, we aim to quantify the impact of this parameter on overall performance.

Our findings indicate a strong linear correlation between thread count and execution time. As the number of threads increases, the total execution time also increases proportionally. This suggests that the overhead associated with thread management and synchronization significantly contributes to the overall execution cost.

The observed relationship can be attributed to increased contention among threads competing for shared resources and the overhead introduced by thread creation, context switching, and synchronization mechanisms. When threads are forced to wait for longer periods to access shared resources or synchronize with other threads, they are less likely to make productive use of their allocated CPU time, leading to decreased system throughput.

These results underscore the importance of carefully considering thread count when designing and optimizing multithreaded applications. By identifying the optimal number of threads for a given workload and employing strategies to minimize contention and synchronization overhead, developers can effectively harness the benefits of parallelism and improve application performance.

## A. Approach for Contention Minimization

We have realized that synchronization of threads minimises the contention probability. We now explore new ideas for further reduction of contention probability. The threads looking to access memory segment sense whether the desired memory segment is locked by a thread. If yes, they wait for a random amount of try and try to access the memory. It can be reasoned that with this approach contention probability will be minimised.

In multi-threaded environments, thread synchronization plays a crucial role in minimizing contention when multiple threads attempt to access shared resources, such as memory segments. We have observed that effective synchronization techniques can significantly reduce the probability of contention, thereby enhancing overall system performance. However, there is still potential for further reducing contention probability through innovative approaches.

To advance our contention minimization strategy, we propose a mechanism whereby threads attempting to access a shared memory segment first ascertain whether that segment is currently locked by another thread. This can be achieved through a lightweight status check or flagging mechanism that indicates the lock state of the memory segment. If a thread detects that the desired memory segment is locked, it does not immediately enter a blocking state. Instead, it will implement a randomized backoff strategy.

The randomized backoff mechanism entails the following steps:

1) **Lock Status Check**: When a thread identifies that its desired memory segment is locked, it does not immediately wait in a queue. Instead, it assesses the lock state first to avoid unnecessary blocking.
2) **Random Wait Time**: Upon confirming the segment is locked, the thread will wait for a random amount of time before attempting to access the memory segment again. This time interval can be calculated using a uniform random distribution within a defined range. The randomness introduces variability in the timing of subsequent access attempts among competing threads, which helps to alleviate the bottleneck caused by multiple threads trying to access the same memory segment simultaneously.
3) **Re-attempt Access**: After the random wait period, the thread will again check the lock status of the memory segment. If it finds the segment still locked, it repeats

the randomized waiting process. If the memory segment is available, the thread can then proceed to access it.

4) **Adaptive Backoff**: To enhance the effectiveness of this approach, the random wait time can be adjusted dynamically based on the contention level. For instance, if a thread experiences repeated failures in acquiring the lock, it may increase its wait time, thereby reducing the likelihood of repeated contention with other threads. Conversely, if the segment becomes frequently available, the thread may decrease its wait time to improve throughput.

This approach leverages the principles of probability and randomness to minimize contention effectively. By allowing threads to stagger their attempts to acquire locks, we can significantly reduce the likelihood of simultaneous access requests that lead to contention. Consequently, this method not only decreases the wait time for threads trying to access locked memory segments but also improves overall system efficiency by ensuring that threads are not idle while waiting for access.

The proposed approach for contention minimization focuses on adaptive synchronization through random backoff strategies, which offer a promising avenue for enhancing performance in multi-threaded environments. By intelligently managing access attempts, we can further optimize resource utilization and improve responsiveness in systems where contention is a significant concern.

## B. Improvements to the Stochastic Model

It is realistic to consider the model in which if a memory segment is accessed, it is highly like that the adjacent memory segments are also needed by the thread and they will be accessed. It is reasonable to assume Markov dependence between accessed segments. We are currently developing the improved stochastic model and associated performance evaluation.

In the context of multi-threaded applications, understanding and predicting memory access patterns is crucial for optimizing performance and reducing contention. Traditionally, stochastic models have been used to analyze memory access behaviors, but we recognize the need to enhance these models to better reflect real-world scenarios. One significant improvement is to incorporate the concept of spatial locality, particularly the Markov dependence between accessed memory segments.

*1) Rationale for Markov Dependence:* When a thread accesses a specific memory segment, it is not merely an isolated event. Empirical observations indicate that threads often require not only the primary segment they are accessing but also adjacent memory segments. This phenomenon is known as spatial locality, which refers to the tendency of threads to access memory locations that are physically close to one another within a short time frame.

To model this behavior accurately, we propose integrating a Markov dependence assumption into our stochastic model. In this framework, the access pattern of a thread can be viewed as a Markov process, where the probability of accessing a particular memory segment depends on the segments that have been accessed in the recent past. This dependency can be represented mathematically as follows:

$$P(X_n|X_{n-1}, X_{n-2}, \ldots, X_{n-k}) = P(X_n|X_{n-1}), \quad (1)$$

where $X_n$ represents the memory segment accessed at time $n$, and the condition indicates that the future access depends only on the most recent access. This simplification captures the essential nature of memory access patterns while enabling more efficient computations.

*2) Development of the Improved Stochastic Model:* In developing our improved stochastic model, we aim to capture the following aspects:

1) **State Representation**: Each state in our model corresponds to a specific memory segment, and the transitions between states reflect the probability of accessing neighboring segments after a given segment has been accessed. By defining transition probabilities based on historical access patterns, we can create a more accurate representation of how threads navigate memory.
2) **Transition Probabilities**: The transition probabilities can be estimated using real access logs or simulated environments. By analyzing historical data, we can derive the likelihood of moving from one memory segment to adjacent segments. This empirical approach ensures that our model aligns closely with actual thread behaviors.
3) **Performance Evaluation**: Alongside the model's development, we are focusing on performance evaluation to quantify the improvements brought about by incorporating Markov dependence. We will conduct a series of experiments to compare the performance of traditional models against our improved stochastic model. Key performance metrics will include average access latency, contention rates, and overall throughput.
4) **Simulation Framework**: To facilitate the testing of our model, we will establish a robust simulation framework that allows us to manipulate various parameters, such as the degree of locality and contention levels, in order to observe their effects on memory access patterns. This will help us to validate the accuracy and reliability of our improved model in various scenarios.

*3) Implications for System Performance:* By refining our stochastic model to account for Markov dependence among memory accesses, we anticipate several positive outcomes. First, we expect reduced contention due to better predictions of memory segment requirements, leading to fewer simultaneous access attempts on adjacent segments. Second, our enhanced understanding of access patterns may inform better resource allocation strategies, allowing the system to preemptively load relevant memory segments into cache, thereby improving data access times.

The improvements to our stochastic model will provide a more nuanced understanding of memory access behavior in multi-threaded environments. By incorporating Markov dependence and evaluating the model's performance, we aim to contribute to more efficient memory management strategies, ultimately enhancing system performance and responsiveness.

## IV. Conclusion

The stochastic model developed in this work offers a detailed analysis of memory access contention in a multi-threaded system. Our results show that as the number of threads increases, the likelihood of contention rises rapidly, leading to exponentially increasing delays in memory access. For low thread counts, contention is minimal, but with larger thread numbers, performance degradation becomes severe. This underscores the need for efficient memory management and synchronization techniques, especially in systems with high concurrency. Future work may explore more sophisticated synchronization methods or memory management algorithms to mitigate contention and improve performance in highly parallel systems.

## References

[1] C. Breshears, The Art of Concurrency: A Thread Monkeys Guide to Writing Parallel Applications, Sebastopol, CA, USA:OReilly Media, 2009.

[2] Scott, Michael & Brown, Trevor. (2024). Shared-Memory Synchronization. 10.1007/978-3-031-38684-8.

[3] Sherry Sahebi, Mengfan Yao, Siqian Zhao, and Reza Feyzi Behnagh. 2024. MoMENt: Marked Point Processes with Memory-Enhanced Neural Networks for User Activity Modeling. ACM Trans. Knowl. Discov. Data 18, 6, Article 155 (July 2024), 32 pages. https://doi.org/10.1145/3649504

[4] Goetz, Brian & Peierls, Tim & Bloch, Joshua & Bowbeer, Joseph & Holmes, David & Lea, Doug. (2006). Java Concurrency in Practice.

[5] Vijayalakshmi Saravanan, S. Kaushik, P. Sai Krishna, and D. P. Kothari. 2013. Performance analysis of multi-threaded multi-core CPUs. In Proceedings of the First International Workshop on Many-core Embedded Systems (MES '13). Association for Computing Machinery, New York, NY, USA, 49–53. https://doi.org/10.1145/2489068.2489076

[6] Michael, M.M. (2010). Memory Management in Concurrent Algorithms. In: Touili, T., Cook, B., Jackson, P. (eds) Computer Aided Verification. CAV 2010. Lecture Notes in Computer Science, vol 6174. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-14295-6_4

[7] Xin Wei, Liang Ma, Huizhen Zhang, Yong Liu, Multi-core-, multi-thread-based optimization algorithm for large-scale traveling salesman problem, Alexandria Engineering Journal, Volume 60, Issue 1, 2021, Pages 189-197, ISSN 1110-0168, https://doi.org/10.1016/j.aej.2020.06.055.

[8] Gepner, Pawel & Kowalik, Michal. (2006). Multi-Core Processors: New Way to Achieve High System Performance. PARELEC 2006 - Proceedings: International Symposium on Parallel Computing in Electrical Engineering. 9-13. 10.1109/PARELEC.2006.54.

[9] Saugata Ghose. General-Purpose Multicore Architectures. 2024. arXiv:2408.12999. Available online: https://arxiv.org/abs/2408.12999.

[10] Minwen Ji, Edward W. Felten, and Kai Li. 1998. Performance measurements for multithreaded programs. SIGMETRICS Perform. Eval. Rev. 26, 1 (June 1998), 161–170. https://doi.org/10.1145/277858.277900