International Journal of Computer Applications (0975 - 8887)
Volume 186 - No.64, January 2025

Code and Performance-based Metrics for Multithreaded Object-Oriented Software

R. Gururaj, PhD
BITS Pilani, Hyderabad Campus

Tarimala Vignesh Reddy
BITS Pilani, Hyderabad Campus

Revanth Nalla
BITS Pilani, Hyderabad Campus

ABSTRACT

In the era of multicore processors and distributed systems, the efficient design and performance evaluation of multithreaded programs have become crucial for achieving scalable, high-performance software. However, traditional metrics for evaluating these programs often fall short of capturing the complex interactions between threads, resources, synchronization mechanisms, and execution flow. They often fail to capture the intricacies in multithreaded environments.
This paper introduces advanced metrics tailored for multithreaded applications. This paper proposes a set of complex metrics that can be used to judge the quality of a multi-threaded program based on both the static code and the program's performance. The aim is to identify both good and bad code practices while coding a multi-threaded program. Using these metrics, developers can get actionable insights into the quality and performance of the code, enabling them to refine their designs and optimize execution for better scalability and reliability.

Keywords

Software Metrics, Object-Oriented Software, Static Metrics, Dynamic Metrics, Multi-threading

1. INTRODUCTION

With the widespread adoption of multicore processors and distributed computing environments, developing complex and scalable multithreaded systems is an important software engineering skill [1, 4]. Multithreaded programming, however, introduces new challenges compared to traditional single-threaded software.
Unlike traditional single-threaded programs, multithreaded systems must carefully manage how threads interact with each other while sharing resources, such as databases or memory. Without proper coordination, these interactions can lead to serious issues [3, 14]. For example, race conditions occur when two threads try to modify the same resource at the same time, leading to unpredictable outcomes. Similarly, deadlocks can happen when threads become stuck waiting for each other indefinitely, essentially freezing the system [1]. Even more subtle problems, like data inconsistencies, arise when simultaneous operations create discrepancies in the information stored or processed [3, 9].
These challenges are not just theoretical, but have real-world implications. A poorly designed multithreaded system can result in performance bottlenecks, crashes, or corrupted data [4]. Worse yet, these issues can be difficult to identify and fix, especially in large-scale systems with many interconnected components. As software engineers strive to build scalable systems, the need for better metrics to assess and manage multiple threads has become increasingly urgent [7, 11].
This research is motivated by the growing demand for such tools. Specifically, it aims to fill a critical gap by developing a comprehensive suite of advanced metrics tailored for multithreaded applications [1, 3]. These metrics will serve as a diagnostic toolkit for developers, helping them uncover inefficiencies and potential problems that might otherwise go unnoticed. By offering clear insights into how threads interact, how resources are utilized, and where bottlenecks occur, these metrics will empower developers to optimize their systems more effectively.
The idea behind this research is to address the gap by developing a suite of advanced metrics for multi-threaded applications. These metrics will reveal inefficiencies and potential issues that impact the functioning of the program and also the ability for developers to understand the code base [8, 9]. The remainder of the paper is organized as follows. In Section 2, we discuss related work, and in Section 3, we elaborate on the proposed metrics. The Section 4 focuses on verification of the proposed concepts, and in Section 5 we provide details of analysis. Finally, the Section 6 gives concluding remarks.

2. LITERATURE REVIEW

Object-oriented metrics have long served as essential tools for assessing software quality in terms of complexity, maintainability, and reliability. Early research into these metrics has emphasized both static and dynamic measures to capture different dimensions of software design and execution. Static metrics, derived from code structures and design documents, typically include parameters such as Lines of Code (LOC), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO). Dynamic metrics, in contrast, focus on runtime behavior, providing insights into how classes and objects interact during actual execution [1, 5, 6, 9, 14]. The work [1] explores combining static and dynamic indicators to form hybrid metrics that offer a more comprehensive evaluation, revealing both design-time and runtime quality attributes.
Over the years, object-oriented metrics have been categorized into several key dimensions, each reflecting specific aspects of software quality [5, 10]:

—**Size Metrics**: Include measures like the number of classes, methods, and LOC, providing a basic scale of the system [6].

—**Complexity Metrics**: Assess class hierarchies, control flow intricacies, and polymorphic behavior, giving a sense of how challenging the code may be to understand or maintain [9].

—**Coupling Metrics**: Evaluate the degree of interdependence between classes. High coupling often indicates that changes in one class can propagate widely, reducing maintainability [14].

—**Cohesion Metrics**: Examine how well the internal elements of a class work together. Low cohesion suggests scattered logic that can hinder comprehension and maintenance [1, 9].

—**Inheritance Metrics**: Look at how effectively classes reuse attributes and methods through inheritance. Deeper inheritance trees can suggest better reuse but may also increase complexity [2, 7].

Several well-known metric suites have emerged from these categories. In [14], the CK metrics, introduced by Chidamber and Kemerer, are widely cited for their empirical correlations with software defects and maintenance effort. Studies applying CK metrics have found that while larger classes or classes with more complex methods may be prone to defects, not all metrics yield consistent linear relationships with defect counts [3]. For instance:

—Larger class size often correlates with increased defects.

—Weighted methods per class (WMC) and coupling between objects (CBO) may not always show strong linear correlations with defect density.

Another set of metrics known as MOOD (Metrics for Object-Oriented Design) focuses on encapsulation, inheritance, coupling, and polymorphism introduced in [2]. Empirical evaluations highlight:

—**Method Hiding Factor (MHF)** and **Attribute Hiding Factor (AHF)**: Higher values indicate better encapsulation and thus greater modularity.

—**Method Inheritance Factor (MIF)** and **Attribute Inheritance Factor (AIF)**: Larger values suggest effective code reuse and potentially simpler extension mechanisms.

—**Coupling Factor (CF)**: Lower values denote looser inter-class dependencies, making the system easier to maintain.

—**Polymorphism Factor (PF)**: Higher values indicate greater use of polymorphism, which can enhance system adaptability and extensibility as mentioned in [2, 7].

From [5, 10, 12], it can be derived that despite these advancements, researchers have encountered difficulties in defining universal thresholds and interpretations for such metrics. Quality norms may differ based on application domains, project sizes, or development methodologies, making it challenging to apply one-size-fits-all benchmarks. Moreover, some studies [3] emphasize the complexity of directly linking certain metrics to defects, suggesting that non-linear relationships or transformations might be necessary .

As software evolves, especially with the rise of multicore processors and distributed systems, traditional object-oriented metrics—initially conceived for single-threaded, sequential programs—fall short in capturing the complexities introduced by concurrency [1, 4, 11, 13]. Issues like thread contention, race conditions, deadlocks, and synchronization overheads arise not from static structures but from dynamic interactions between concurrently executing threads. These concurrency-specific challenges demand metrics that can:

—Assess synchronization overhead and waiting time.

—Measure the load imbalance caused by uneven work distribution among threads.

—Evaluate thread contention probabilities for shared resources.

—Identify potential deadlocks or starvation scenarios.

The need to adapt or create new metrics tailored for multithreaded programs is becoming increasingly critical as parallelism becomes the norm rather than the exception [8, 9, 13]. Integrating concurrency-aware metrics with traditional object-oriented measures would provide a more holistic toolkit, enabling developers to diagnose inefficiencies, optimize execution paths, and enhance both the scalability and maintainability of modern software systems. In essence, while classical object-oriented metrics lay the groundwork for evaluating structure and maintainability, expanding them to consider concurrency opens new avenues for understanding software quality in the face of complex, parallel execution environments.

3. PROPOSED NEW METRICS FOR PROGRAM EVALUATION

The metrics proposed as part of this paper can be divided into two categories:

—*Static Metrics*: These are code quality metrics which can help analyse the multithreaded program based on its static code. It mostly focuses on the good coding practices that are followed.

—*Dynamic Metrics*: These are metrics that are calculated at runtime using a profiler and analyse the performance of the multithreaded program based on factors like its CPU utilisation, usage of other resources, deadlocks, starvation etc.

3.1 Static Metrics/Code Quality Metrics

3.1.1 Thread Coupling Factor. This metrics defines the degree to which classes are interdependent through threading constructs on classes which are not responsible for the creation of those threads. It can be given by the following formula:

$$\text{TCF} = \frac{\text{Number of Inter-class Thread Interactions}}{\text{Total number of possible Inter-class Interactions}}$$

A high Value of TCF can indicate that thread management is handled across multiple classes which can lead to more deadlocks and an overall bad-quality program. On the other hand, a low value can indicate more independent classes which promotes better encapsulation and less requirement for testing.

3.1.2 Lines of Code per Critical Section. It measures the average number of lines per critical section of the program. Critical sections are parts of code that require synchronization and primarily include regions which deal with database access etc. Long critical sections indicate complex synchronization, which increases the difficulty of reading and understanding the code. A high value of LOC-CS suggests that critical sections are too large leading to more complicated interactions between threads. This makes the code hard to follow. It can be calculated by the following formula:

$$\text{LOC-CS} = \frac{\text{Total Lines of Code in Critical Sections}}{\text{Number of critical sections}}$$

3.1.3 Synchronous Call Frequency. This metric measures the proportion of synchronous calls in a multithreaded program. It can be calculated as the following:

$$\text{SCF} = \frac{\text{Number of Synchronous Calls}}{\text{Total number of Function calls}}$$

A synchronous call forces a thread to wait for another thread to complete a task, introducing potential bottlenecks or blocking behaviour. A high SCF suggests that the program has frequent points where threads are blocked waiting for each other, leading to more complex flow control and reduced readability. Reducing synchronous calls where appropriate can enhance both performance and readability.

3.1.4 Concurrency Complexity Metric. This metric measures the complexity introduced by concurrent threads and their critical sections. It is based on the intuition that main defects in code arise when a large number of threads compete for a critical section. Therefore the quality of the program depends on the number of threads competing for the critical section, not just based on the length of the critical section.
It is calculated as the following:

$$\sum_{i=1}^{n} T_i \times L_i$$

where T_i = Number of threads in the ith critical section and L_i represents the number of Lines of code in the ith critical section.
The larger the critical section or the more threads interacting with it, the harder the code is to read and maintain. High CCM values indicate more complicated thread interaction, which increases the difficulty of understanding and managing the code. Large or numerous critical sections usually reduce the readability of the code due to the intricacies of synchronisation and potential contention.

3.1.5 Thread Contention Probability. In multithreaded programs it is important to understand how likely it is that threads will contend for critical sections. High contention probability can lead to performance bottlenecks. The goal of the metric is to predict the chances of thread contention based on the number of threads accessing critical sections and the size of those sections. It can be calculated as follows:

$$\text{TCP} = \sum_{\text{CS}} \left(\frac{\text{NTA}}{\text{TT}} \times \frac{\text{SCS}}{\text{TCS}} \right) \tag{1}$$

where TCP is the Thread Contention Probability, CS is the Critical Sections, NTA is the Number of Threads Accessing the Critical Section, TT is the Total Threads, SCS is the Size of the Critical Section, and TCS is the Total Code Size.

3.2 Dynamic/Performance metrics

3.2.1 Parallelism Overhead. The metric measures the proportion of program execution time spent in synchronization, for example, acquiring and releasing locks. It is calculated as the following:

$$\text{PO} = \frac{\text{Time spent in synchronization}}{\text{Total Program Execution time}}$$

High overhead can indicate inefficient use of threads or poor synchronization strategies, which often correlates with difficult-to-read code. A high PO suggests that the program is spending too much time on synchronization instead of executing useful work. This can often point to complicated synchronization logic that makes the code harder to follow.

3.2.2 Thread Starvation Rate. It is a metric that measures the amount of time threads are unable to proceed because they are waiting for resources or conditions to be met. This is prevalent in programs that use a large number of threads run on single-core systems where threads wait for CPU allocation for a significant amount of time relative to the amount of useful work they perform. It is calculated as:

$$\text{TSR} = \frac{\text{Time thread is starved}}{\text{Total thread execution time}} \times 100$$

A high thread starvation rate indicates poor resource management or contention, where threads spend excessive time waiting rather than executing tasks. Avoiding thread starvation by using better scheduling algorithms or load balancing ensures that the thread progresses and the program remains responsive.

3.2.3 Lock Competitiveness Index. When multiple threads frequently contend for locks, it can create bottlenecks, and deadlocks and reduce parallelism. This metric quantifies the level of lock competition by comparing the number of times threads attempt to acquire a lock to the number of successful acquisitions, weighted by the size of the critical section. Values close to 0 indicate that every lock acquisition was correct in the first few attempts which implies that the code is very efficient. It is calculated as follows:

$$\text{LCI} = \sum_{\text{L}} \left(\frac{\text{FLA}}{\text{TLA}} \times \frac{\text{SCS}}{\text{TCS}} \right) \tag{2}$$

Where: LCI is the Lock Contention Index, L is the Locks, FLA is the Failed Lock Acquisitions, TLA is the Total Lock Attempts, SCS is the Size of the Critical Section, and TCS is the Total Code Size.

3.2.4 Load Imbalance. it measures the workload distribution among threads. It is calculated as:

$$\text{LI} = \frac{\text{Max-Thread execution time} - \text{Min-Thread execution time}}{\text{Max Thread execution time}}$$

3.2.5 Thread Resource Utilization Coefficient. This metric evaluates how well threads are at utilising the resources that they acquire such as CPU, memory, IO etc. It is calculated as:

$$\text{TRUC} = \frac{\sum_{\text{R}} \left(\frac{\text{TRU}}{\text{TRH}} \right)}{\text{TNR}} \tag{3}$$

Where: TRUC is the Thread Resource Utilization Coefficient, R is the Resources, TRU is the Time Resource is Actively Used, TRH is the Time Resource is Held by Thread, and TNR is the Total Number of Resources.
Values near 1 indicate optimal resource usage, while values less than half suggest inefficient resource utilisation i.e, resource are held for significantly longer time than they are used.
A high Load imbalance(close to 1) means that some threads are overworked while others are underutilized, leading to inefficient resource use. Proper workload distribution, improving overall, performance and reducing idle time.

4. VERIFICATION

In order to verify if the experimental outcomes of the metric computation align with the intuitive reasoning behind the development of the metric, common multithreading use cases in Java were used to compute the metrics. Two programs which implement a producer-consumer problem in two different methodologies were used, Program A uses synchronization methods to implement the problem, while Program B uses Java Blocking queue to implement the same. The comparative analysis of the metrics on Programs A and B are plotted in Fig 1, Fig 2 and also tabulated in Table 1 and Table 2.

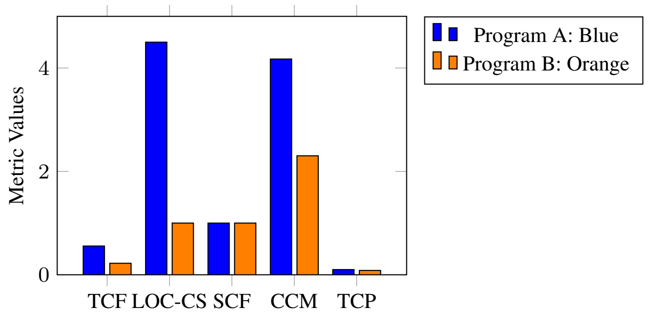

Fig. 1: Static Metric Comparison for Programs A and B

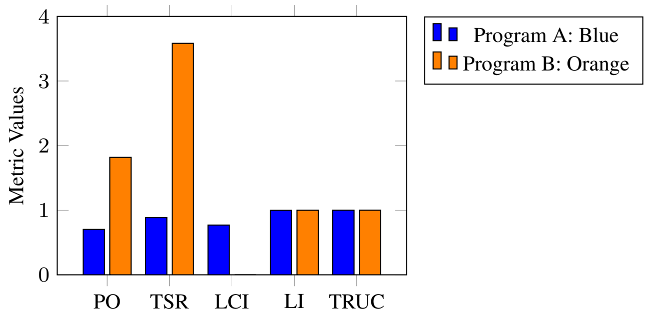

Fig. 2: Dynamic Metric Comparison for Programs A and B

Static Metric	Program A	Program B
TCF	0.556	0.222
LOC-CS	4.5	1
SCF	1	1
CCM	4.174	2.302
TCP	0.1	0.0833

Table 1. : Static Metrics for Programs A and B

Lower TCF in Program B indicates less inter-class thread interaction, implying better modularity, which is as expected as Java Blocking queue prevents any unnecessary thread interaction between the producer and the consumer. Also Program B has a smaller critical section making it more readable. Program B's lower concurrency complexity (CCM) suggests simpler and more maintainable code. Lower TCP in Program B indicates lower contention probability, supporting better concurrency. So the overall conclusion from the static metric analysis is that Program B is relatively simple to read, and understand and overall more maintainable code which concurs with the fact that it was written with more abstract data structures compared to Program A.
Program B's lower parallelism overhead suggests more efficient use of synchronization. Program B has less thread starvation, suggesting better resource scheduling and fewer blocking delays. Also it's lack of lock contention implies reduced chances of deadlock and better resource access. Both programs have nearly identical load imbalance, indicating similar workload distribution among threads. Both programs have high resource utilization, suggesting that each utilizes its acquired resources effectively.
Another set of 2 programs C and D were chosen which perform Linked List operations. In program C, a single lock is applied over the entire list while in program D each node had its own lock. Metrics were calculated both manually in the case of the static metrics and Java utility libraries to calculate the dynamic metrics. The results are displayed in Fig 3, Fig 4 and Table 3 and Table 4

Dynamic Metric	Program A	Program B
PO	0.703	1.818
TSR	0.886	3.583
LCI	0.769	0
LI	0.998	0.999
TRUC	0.9991	0.9996

Table 2. : Graphical analysis of the Dynamic metrics

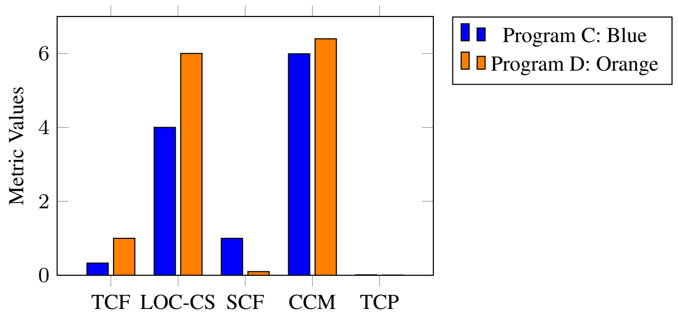

Fig. 3: Static Metric Comparison for Programs C and D

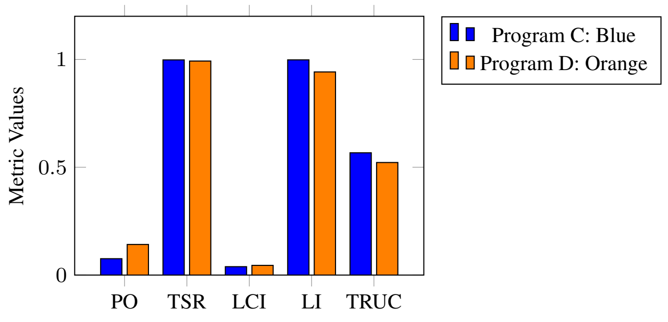

Fig. 4: Dynamic Metric Comparison for Programs C and D

One of the notable metrics here to be observed is the TCP. which is very low in Program D as the same number of threads are contending for more number of locks i.e for each node. But in case of Program C, all threads are competing for the same single lock over the entire linked list. Thus it can be experimentally concluded that Program C is likely to face more contention compared to D
Also, another set of two programs, E and F, were designed to evaluate multithreaded file-writing operations. In Program E, synchronized methods were used to control thread access to a shared file resource, while in Program F, a java.util.concurrent ReentrantReadWriteLock was employed to manage thread access more efficiently. The verification process concluded that Program F is superior in terms of readability, maintainability, and concurrency efficiency. The use of ReentrantReadWriteLock in Program F resulted in reduced contention and better scheduling of threads, aligning with the intuitive reasoning behind its design. Program F's dynamic metrics

Static Metric	Program C	Program D
TCF	0.33	1
LOC-CS	4	6
SCF	1	0.1
CCM(ln)	5.991	6.397
TCP	0.012	0.00033

Table 3. : Graphical analysis of the static metrics

Dynamic Metric	Program C	Program D
PO	0.076	0.142
TSR	0.9978	0.9925
LCI	0.039	0.045
LI	0.998	0.942
TRUC	0.567	0.522

Table 4. : Graphical analysis of the Dynamic metrics

highlighted its ability to handle multithreaded operations more effectively than the synchronized approach of Program E. This confirms that Program F offers a more modular and scalable solution.

5. ANALYSIS

5.1 Thread Coupling Factor (TCF)

It helps identify interdependencies across classes involving threading constructs, thus allowing developers to analyse the modularity of thread management. A low value of TCF, signifies better encapsulation, reducing the likelihood of deadlocks and improving testability. However a drawback of the metric is that low value may not always translate to better code design if critical interdependencies are necessary for the codebase.
A low value of TCF(close to 0), indicates minimal coupling, that thread handling is isolated to specific classes, suggesting better modularity and encapsulation. High values suggest heavy coupling across classes, increasing the likelihood of potential deadlocks.

5.2 Lines of Code Per Critical Section (LOC-CS)

it provides insights into the complexity of critical sections in a program and therefore identifies areas prone to high synchronization overhead. However, some of the areas overlooked by this metric is that it overlooks the complexity inside the critical sections taking size as the only factor affecting the complexity. Therefore if the program logic in the critical section is too complex a low LOCS-CS may not guarantee better code design.
A value close to 1, suggests a small critical section, reducing thread contention and synchronization overhead. Higher values usually indicate larger sections, which can lead to complex inter-thread dependencies and make the code harder to understand and maintain.

5.3 Synchronous Call Frequency (SCF)

The metric identifies potential bottlenecks due to synchronous calls, indicating where asynchronous alternatives might improve performance. It helps assess potential blocking behaviour in the code, which is crucial for optimizing response times. On the other hand, a lower SCF can't be always tied to better code design as some applications need synchronous calls for application correctness.
Low SCF(close to 0) suggests minimal blocking, indicating that the threads can operate more independently. High SCF indicates more blocking in the code which can also lead to potential deadlocks if there is coupling to other classes with high SCF as well.

5.4 Concurrency Complexity Metric (CCM)

The metric directly correlates with the complexity of concurrent interactions, highlighting areas where many threads interact with critical sections taking into account not just the size of the critical section but the number of threads that participate in interaction with the critical section. The only drawback of the metric is that it does not account for the nature of the thread interactions just the volume. A low CCM(close to 0) indicates more simple and maintainable code with fewer complex interactions among threads. A high CCM, suggests more complexity which can reduce the maintainability and the readability of the multithreaded program.

5.5 Thread Contention Probability (TCP)

the metric predicts how likely threads are to contend for resources, allowing developers to identify potential bottlenecks by looking at the code. However, since it is a static metric this only measures the probability while actual contention may vary depending on runtime conditions.
Low TCP(close to 0) indicates a low likelihood of contention, suggesting that threads can operate with minimal interference.

5.6 Parallelism Overhead (PO)

It directly quantifies the cost of synchronization logic itself, indicating the amount of program time that was not useful as it was spent in synchronization. It is useful for developers to identify the inefficiencies in parallelism.
Low value of PO indicates minimal overhead, with more time spent on actual work than synchronization whereas a high value basically reduces the effective parallelism.

5.7 Thread Starvation Rate (TSR)

It measures how effectively resources are allocated among threads, highlighting if some of the threads are starved and thus is useful for identifying resource management issues. However since is a very time-dependent metric, it can be observed that the values can be significantly impacted by the hardware and even in different executions on the same hardware.
A low value of TSR(close to 0), suggests efficient resource allocation with minimal starvation time for threads.

5.8 Lock Competitiveness Index (LCI)

It quantifies how frequently the threads try to acquire locks over a critical section and fail as it is massive performance bottleneck. One of the noticeable drawbacks of this metric is that it does not account for the importance of each individual lock and treats them all equally. Also, asymptotically it can be observed intuitively that for highly concurrent applications a high LCI is unavoidable.
A low LCI value suggests minimal lock contention and therefore represents the ideal case where most threads are able to acquire locks in few attempts. High LCI is an indirect indication of potential deadlocks and poor performance.

5.9 Load Imbalance (LI)

the metric offers a direct measure of the workload distribution among threads, essential for ensuring the efficient use of resources. Helps in optimizing task allocation to minimize idle time for each

thread and maximize the CPU utilization. A drawback is that it does not account for the difference in task complexity for each thread as that is hard to quantify.
Low values, close to 0 indicate balanced workload distribution, maximizing resources. a high value indicates some threads are overworked while some are underutilized.

5.10 Thread Resource Utilization Coefficient (TRUC)

it measures the efficiency with which threads utilize acquired resources, providing insights into potential resource wastage. It is thus useful to identify areas where resources are held longer than needed. High TRUC although mostly indicates the above, may not always be bad if resources are utilized effectively.
TRUC close to 1 indicates optimal resource usage with minimal waste. Values less than 0.5, suggest inefficient resource utilization, where resources are held without productive work.

6. CONCLUSION

In conclusion, this paper introduces a novel set of metrics designed specifically aimed at evaluating and optimizing multi-threaded programs. Traditional metrics often fail to capture the nuanced interactions between threads, resources and synchronization overheads that are unique to concurrent systems.
The analysis demonstrates that these metrics provide actionable insights into critical aspects of multithreading, including resource allocation efficiency, synchronization overhead, load balancing, and potential for thread starvation and deadlock. These metrics also reveal opportunities for optimization that would otherwise be challenging to identify with existing metrics.
Further research could involve validating these metrics across diverse multithreaded environments (e.g., high-performance computing, distributed databases, real-time applications) and exploring automation in collecting and analyzing these metrics using advanced profiling tools. These metrics can also be incorporated into development tools, making them accessible to developers aiming to create efficient, scalable multithreaded applications. These metrics help developers fine-tune multithreaded software, making it faster, more scalable, and more reliable.

7. REFERENCES

[1] Ponnala, Ramesh & Reddy, Dr. (2019). Object Oriented Dynamic Metrics in Software Development: A Literature Review.

[2] R. Harrison, S. J. Counsell and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," in *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491–496, June 1998, doi: 10.1109/32.689404. keywords: Mood;Software metrics;Encapsulation;Software measurement;Computer Society;Application software;Software quality;Emotion recognition;Software systems;Project management.

[3] Suresh, Yeresime & Pati, Jayadeep & Rath, Santanu. (2012). Effectiveness of Software Metrics for Object-oriented System. *Procedia Technology*, 6, 420–427. https://doi.org/10.1016/j.protcy.2012.10.050

[4] VanderWiel, S. P., Nathanson, D., & Lilja, D. J. (1996). Complexity and performance in parallel programming languages. *University of Minnesota High-Performance Parallel Computing Research Group Technical Report* HPPC-96-02.

[5] L. Tahvildari and A. Singh, "Categorization of object-oriented software metrics," *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era (Cat. No.00TH8492)*, Halifax, NS, Canada, 2000, pp. 235–239 vol.1, doi: 10.1109/CCECE.2000.849705. keywords: Software metrics;Software measurement;Size measurement;Object oriented modeling;Software design;Documentation;Programming;Software engineering;Area measurement;Program processors

[6] Lorenz, M. & Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., USA.

[7] Lanza, M., & Marinescu, R. (2007). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Germany: Physica-Verlag.

[8] Yeresime Suresh, Jayadeep Pati, Santanu Ku Rath, Effectiveness of Software Metrics for Object-oriented System, *Procedia Technology*, Volume 6, 2012, Pages 420–427, ISSN 2212-0173, https://doi.org/10.1016/j.protcy.2012.10.050

[9] Wei Li, Sallie Henry, Object-oriented metrics that predict maintainability, *Journal of Systems and Software*, Volume 23, Issue 2, 1993, Pages 111–122, ISSN 0164-1212, https://doi.org/10.1016/0164-1212(93)90077-B

[10] Rajender Singh Chhillar and Sonal Gahlot. 2017. An Evolution of Software Metrics: A Review. In *Proceedings of the International Conference on Advances in Image Processing (ICAIP '17)*, Association for Computing Machinery, New York, NY, USA, 139–143. https://doi.org/10.1145/3133264.3133297

[11] Mei-Huei Tang, Ming-Hung Kao and Mei-Hwa Chen, "An empirical study on object-oriented metrics," *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*, Boca Raton, FL, USA, 1999, pp. 242–249, doi: 10.1109/METRIC.1999.809745.

[12] El-Emam, K. (2002). "Object-oriented metrics: A review of theory and practice." *Advances in Software Engineering: Comprehension, Evaluation, and Evolution*, 23–50.

[13] Li, Wei, and Sallie Henry. (1993). "Maintenance metrics for the object oriented paradigm." *[1993] Proceedings First International Software Metrics Symposium*. IEEE.

[14] Chidamber, Shyam R., and Chris F. Kemerer. (1991). "Towards a metrics suite for object oriented design." *Conference proceedings on Object-oriented programming systems, languages, and applications.*