A Survey on Dynamic Thread Pool Management Techniques

Abhigna Bheemineni, Nishka Sathisha, Swetha Chiliveri

ABSTRACT

Dynamic Thread Pool Management (DTPM) techniques are essential for optimizing the performance and resource utilization of concurrent and parallel computing systems. This survey paper categorizes existing DTPM techniques and examines their characteristics, advantages, and limitations. It starts with an overview of thread pools' importance and the need for dynamic management techniques to handle varying workloads efficiently. The techniques are classified into Load-based, Adaptive, and Predictive categories, each addressing different aspects of dynamic thread pool management. Hybrid Approaches that combine multiple techniques are also discussed. The paper provides insights into the principles and practical implications of DTPM techniques, identifying open research challenges and future exploration avenues.

Additionally, the paper introduces a methodology for evaluating resource manager (RM) configurations using model fuzzing to address the complexity of configuring RMs accurately. It proposes a Distributed Thread Pool (DTP) system for dynamic optimization and overload management, which effectively detects and resolves overload conditions. The paper also discusses related work on dynamic optimization and overload management of Thread Pool Systems (TPS) and presents mathematical models and dynamic adjustment mechanisms for thread pool management.

Finally, the survey delves into three research papers focusing on thread optimization techniques, performance studies, and dynamic optimization designs for thread pool systems. Each paper contributes valuable insights to the realm of thread optimization, providing methodologies to enhance the efficiency of parallel execution and resource usage in computing environments.

1. INTRODUCTION

Dynamic thread pool management, crucial for optimizing performance and resource utilization in concurrent and parallel computing systems, involves adjusting thread pool size and configuration based on varying workloads and system conditions. Thread pools are commonly used in software applications to manage a pool of worker threads that execute tasks asynchronously. Here are some key aspects of dynamic thread pool management: Thread Pool Size Adjustment, Load Balancing, Task Prioritization, Thread Pool Shrinkage, Dynamic Configuration, Monitoring and Metrics, and Concurrency Control. Dynamic thread pool management is particularly beneficial in applications with fluctuating workloads or where resource utilization needs to be optimized dynamically, improving system responsiveness, scalability, and efficiency by adjusting thread pool resources in real-time to match the workload demands.

.NET itself is not a resource manager in the traditional sense. However, it provides a wide range of features and libraries that enable developers to manage various types of resources efficiently within their applications. The paper highlights the complexity of configuring RMs, citing the challenge of evaluating many interdependent configuration parameters and the impracticality of collecting a vast amount of data using performance benchmarks. The proposed model fuzzing technique, combined with a robust system model, provides an efficient and accurate means of evaluating RM configurations, offering substantial computational efficiencies and significant improvements in system performance. The paper reviews the increasing demand for scalable and performance-efficient services due to the growing internet traffic, particularly for distributed systems handling heavy workloads, and discusses the challenges related to middleware services for distributed systems, presenting a detailed comparison between Thread Pool System (TPS) and event-driven models. The TPS is highlighted as a more solid and efficient option for concurrent servers.

The paper discusses the importance of effective thread pool management in multithreading systems to minimize response time and maximize resource utilization. It presents a novel trendy exponential moving average (TEMA) scheme for predicting the number of threads and a prediction-based thread pool management scheme that dynamically adjusts the idle timeout period and thread pool size to adapt to changing environments, evaluating the effectiveness of the proposed approach in terms of response time and CPU usage, comparing it to existing prediction-based schemes and the Sun watermark. In the dynamic field of parallel computing, optimizing thread execution plays a pivotal role in boosting system performance. This survey dives into key research papers exploring the nuances of thread optimization. Beyond abstract summaries, this introduction sets the stage by highlighting the relevance of thread optimization in today's computing landscape.

With the rise of multicore architectures and the increasing need for efficient resource use, understanding thread-level performance is crucial. Our exploration begins with D. Robsman's "Thread Optimization" (US6477561 B1, 2002), providing foundational insights into various optimization strategies. Moving forward, D. Xu and B. Bode's work, "Performance Study and Dynamic Optimization Design for Thread Pool System" (2004), focuses on dynamic optimization and resource efficiency in thread pool systems. Culminating our journey is "Prediction based Dynamic Thread Pool Scheme for Efficient Resource Usage" (2008) by D. Kang, S. Han, S. Yoo, and S. Park. This paper introduces a forward-thinking approach, incorporating prediction mechanisms for dynamic thread pool management and enhanced efficiency. As we explore these papers, the introduction aims to underscore the significance of thread optimization and its practical implications. Subsequent sections will delve into each paper's methodologies and contributions, offering a straightforward understanding of thread optimization in parallel computing.

2. TERMINOLOGY

Dynamic Thread Pool Management, Thread Pool, Load Balancing, Scalability, Resource Utilization, Workload Prediction, Adaptive Resizing, Work-stealing.

3. EXISTING TECHNIQUES

Load-based Techniques:

Fixed-size Thread Pool: Maintains a fixed number of threads in the pool and assigns tasks to available threads. When the pool is full, incoming tasks are queued.

Cached Thread Pool: Dynamically adjusts the size of the thread pool based on the workload. Threads are created as needed and terminated after a specified idle timeout if unused.

Work-stealing Thread Pool: Distributes tasks among worker threads in a hierarchical manner, with idle threads stealing tasks from other threads' queues to maintain workload balance.

Partitioned Thread Pool: Divides the workload into partitions or segments, each managed by a separate pool of threads, enabling better isolation and load distribution.

Adaptive Techniques:

Elastic Thread Pool: Automatically adjusts the size of the thread pool based on workload metrics such as queue length or CPU utilization, scaling up or down as needed to meet demand.

Dynamic Thread Pool Resizing: periodically evaluates system metrics and dynamically adjusts thread pool parameters such as core pool size, maximum pool size, and queue capacity to optimize resource utilization.

Thread Pool Adaptor: Adapts the thread pool configuration based on changes in the application's execution environment or resource availability, ensuring optimal performance under varying conditions.

Feedback-based Adaptation: Collects feedback from the system and adjusts thread pool parameters based on performance metrics or user-defined policies to achieve desired performance goals.

Predictive Techniques:

Workload Prediction-based Resizing: This technique utilizes historical workload data or predictive models to forecast future workload patterns and proactively adjust the thread pool size to accommodate anticipated changes in demand.

Machine Learning-based Adaptation: Applies machine learning algorithms to analyze historical workload patterns and predict future resource requirements, enabling proactive resizing of the thread pool to optimize performance.

Autoregressive Integrated Moving Average (ARIMA): A statistical method used for time-series forecasting, which can be applied to predict future workload trends and adjust thread pool resources accordingly.

Time Series Analysis-based Resizing: Analyzes time-series data of system metrics (e.g., CPU utilization, queue length) to identify patterns and trends, facilitating proactive resizing of the thread pool to align with anticipated workload fluctuations.

These techniques represent a diverse array of approaches to dynamically managing thread pools in concurrent and parallel computing environments, each with its unique characteristics and suitability for specific application scenarios. Hybrid approaches in dynamic thread pool management involve combining multiple techniques or strategies to leverage their respective strengths and address specific requirements more effectively. Here are some examples of hybrid approaches:

Load Prediction with Adaptive Resizing: This approach combines workload prediction techniques with adaptive resizing strategies. Historical workload data or predictive models are used to forecast future workload trends. Based on these predictions, adaptive resizing mechanisms adjust the thread pool size in real-time to proactively accommodate anticipated changes in demand.

Feedback-based Adaptive Load Balancing: Combining feedback-based adaptation with load balancing strategies. Feedback mechanisms continuously collect performance metrics and adjust thread pool parameters based on real-time feedback. Load balancing algorithms distribute tasks among threads to optimize resource utilization, taking into account feedback-driven adjustments to thread pool configuration.

Dynamic Partitioning with Load-based Resizing: Integrating partitioned thread pool techniques with load-based resizing mechanisms. Workload is partitioned into segments, each managed by a separate pool of threads. Load-based resizing dynamically adjusts the size of each partition based on workload metrics, ensuring efficient resource allocation across different segments of the workload.

Machine Learning-guided Hybrid Adaptation: Incorporating machine learning algorithms into hybrid adaptation strategies. Machine learning models analyze historical workload data and system metrics to identify patterns and trends. Based on these insights, hybrid adaptation mechanisms combine multiple thread pool management techniques, dynamically adjusting thread pool parameters to optimize performance and resource utilization.

Combination of Work-stealing and Predictive Techniques: Leveraging both work-stealing thread pool mechanisms and predictive workload forecasting techniques. Work-stealing algorithms dynamically distribute tasks among worker threads to maintain load balance. Predictive techniques forecast future workload trends, enabling proactive adjustments to the thread pool configuration to accommodate anticipated changes in demand.

These hybrid approaches combine the strengths of multiple thread pool management techniques to achieve more robust, adaptive, and efficient resource allocation in concurrent and parallel computing environments. By integrating complementary strategies, they can effectively address the challenges of varying workloads and optimize system performance under dynamic conditions.

Model fuzzing [7] is used to efficiently find performing configurations by combining measurements of RM resource allocations with a model of the managed system. By applying this methodology, the paper reports a 240% increase in throughput compared to a poorly chosen configuration, and significantly reduces the time required for configuration from machine-years to machine-hours.

The paper proposes [9] a distributed thread pool system with an overload management mechanism based on central management and dynamic optimization. The system aims to enhance the performance and scalability of distributed applications by effectively utilizing system resources and maintaining optimal thread pool size. The evaluation results validate the effectiveness of the proposed system in managing overload conditions and optimizing thread pool size based on request rates. In this paper, we have extended DFBTP (Distributed Frequency based Thread Pool) by presenting OCBDTP (Overload Control based Distributed Thread Pool) system comprises a central manager (CM) component and one or more Thread Pool Systems (TPS) distributed across server nodes. System initialization begins with the CM running on the main server, awaiting connections from TPS instances. As each TPS starts on a server node, it connects to the CM, which stores the corresponding server's IP address and initiates a Receiver thread to handle client requests. A round robin scheduler (RRS) within the CM evenly distributes incoming requests among all available TPS instances, facilitating load balancing across the network. Each TPS is responsible for processing received requests independently. The architecture of each TPS involves three detector threads: Request Rate Detector (RRD), Throughput Detector (TD), and Saturation Detector (SD). Upon TPS startup, these threads are initialized to monitor request rates, throughput, and system saturation, respectively. Requests arriving at a TPS are inserted into a request queue and processed by available threads within the thread pool. RRD periodically calculates request frequencies, TD measures throughput by counting and dequeuing responses, while SD adjusts thread pool size based on request rates and monitors for overload conditions.

The paper [8] introduces the Trendy Exponential Moving Average (TEMA) and a novel prediction-based thread pool management scheme to address the challenges mentioned earlier. It begins by discussing the need for managing redundant and lacking threads and highlights the impact of these factors on system performance. The Enhanced Exponential Moving Average (EEMA) scheme, proposed previously, was found to result in inaccuracies and redundancy. To overcome this, the TEMA scheme is introduced, extending the EMA by incorporating the trend of time series to improve prediction accuracy. The evaluation of TEMA against EEMA demonstrates its superiority in terms of accuracy and thread redundancy reduction, despite a slight disadvantage in handling lacking threads. Subsequently, the prediction-based thread pool management scheme is presented, utilizing theorems to calculate the number of new threads required based on the trend of waiting requests. The system dynamically adjusts the number of threads to optimize performance, considering factors like idle threads, worker threads, and new threads. A mathematical model is constructed to calculate the required threads, incorporating predictions from TEMA, and adjusting configuration parameters dynamically. Thresholds for the number of threads and idle timeout periods are proposed, with considerations for various system states to ensure efficient thread management. Finally, the dynamic adjustment of configuration parameters is discussed, focusing on threshold values for the number of threads and idle timeout periods. The system adjusts these parameters based on the current system status to optimize performance. Thread destruction policies are also outlined, ensuring that idle threads are not destroyed under certain conditions to prevent inefficiencies. Overall, the proposed approach aims to optimize thread pool management by accurately predicting thread requirements and dynamically adjusting system parameters in response to changing conditions.

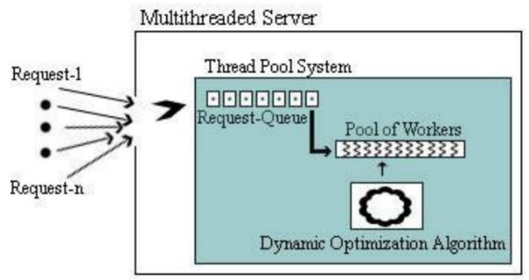

Fig 1: Conceptual Model of Thread Pool System Embedded in a Server.

The overload detection and control mechanism implemented by SD dynamically adjusts thread pool size to mitigate thread context switch and contention overheads. The SD algorithm collects throughput and request frequency data for current and previous phases, then evaluates whether request rates are increasing. If request rates rise, SD adjusts the thread pool size accordingly. Subsequently, SD assesses whether the throughput matches the thread pool size, and if so, considers the pool size appropriate. If throughput fails to improve, indicating an incorrect pool size, SD resets the pool size to the previously saved value. This process of dynamically setting and adjusting pool size helps maintain optimal system performance and resource utilization in response to varying workloads and ensures effective handling of client requests.

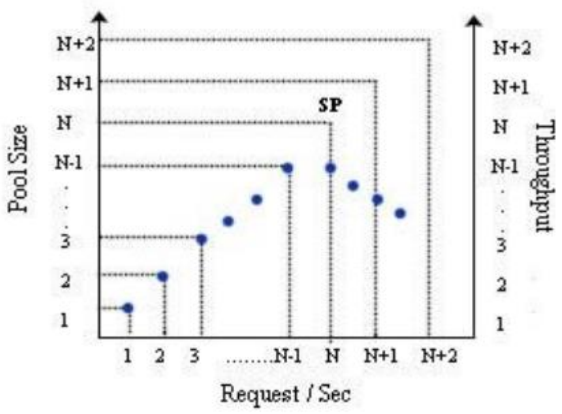

Fig 2: Throughput Starts Falling on Saturation point.

The throughput axis represents the number of requests completed per second, indicated by blue dots. As depicted in the figure, the pool size always remains equal to the request arrival rate, with throughput gradually increasing but eventually decreasing when reaching saturation point. This saturation point, marked by a drop in throughput, occurs when the pool size matches the request arrival rate, yet the system becomes overwhelmed with context switch overhead and thread management tasks. Consequently, system resources are consumed in managing thread context switches and contention rather than performing useful work. The proposed scheme aims to address this saturation point and respond accordingly to optimize system performance.

The approach proposed in [6] to thread pool management, aiming to enhance server response time and system resource utilization. The paper discusses the scheme that is based on the watermark thread pool model, focusing on improving server response time with many user requests and efficiently utilizing system resources with fewer requests. Below are the steps in which thread pool management is carried out:

1. A thread pool is created at the program's inception and dynamically adjusts its size based on the number of user requests.
2. A thread pool watcher is introduced to monitor the current number of threads and predict the expected value for the next period using an exponential average.

3. The main program is responsible for creating the thread pool and waiting for user requests while managing a socket listener.
4. New user registrations trigger the main program to send socket information to the thread pool's requested queue, initiating the processing of user requests.
5. The main program plays a role in checking the status of thread pools.
6. Thread pool is created with a request queue to store user socket information and a worker thread queue to manage threads.
7. Waits for user requests and, upon receiving a user's request, adds the user's socket information to the queue.
8. Wakes up waiting worker threads or creates a new thread if none are waiting, allowing the thread to execute the corresponding request.
9. Each worker thread has waiting and activated states.
10. In the waiting state, the thread deletes itself after a certain amount of time, recording the value by measuring the number of worker threads.
11. Upon activation, the thread measures and records the current number of activated worker threads before executing user requests.
12. Watcher thread calculates the predicted value of required threads for the next stage using exponential averaging, reading the currently recorded number of worker threads periodically.
13. Compares the predicted value with the previously predicted value and creates new threads if the value is larger, ensuring proactive management of thread pools.
14. Criteria for dynamically controlling the number of threads based on user requests are established.
15. Measures include the number of worker threads and a prediction scheme using exponential averaging to consider constant rate changes in previous and current data.

The paper [5] focuses on the design, implementation, and dynamic optimization of a thread pool system using POSIX C and the Pthreads library. The architecture comprises five major modules: Thread queue, Task queue, Thread scheduling, Performance monitoring and adjustment, and Report of system performance. Worker threads and tasks are the central entities, managed through thread and task queues, respectively. The thread pool operates with threads in busy or idle modes, competing for mutex upon receiving task_posted signals. Task submission involves clients submitting tasks, which are placed in the task queue, and worker threads in idle mode competing for mutex to execute tasks. A task dispatcher allows users to submit tasks for execution, placing them in the task queue and notifying waiting worker threads. The paper introduces a dynamicThreadPool algorithm for adjusting the thread pool size based on the average idle time, striving to maintain a balance between throughput maximization and management overhead. Experimental validation on a RedHat Linux system explores the relationship between throughput and thread pool size, considering different workloads. The paper concludes with a comparison between the original static thread pool and the dynamic thread pool, showcasing substantial throughput improvements, particularly for smaller initial pool sizes. The dynamic thread pool's effectiveness in continuously adjusting the pool size towards a stable zone is demonstrated, offering a promising approach to optimize multithreaded application performance.

The technique revolves around managing execution threads in a computer system, specifically in the context of a server application. The goal is to optimize the allocation of threads based on varying processor loads, considering factors such as the nature of the tasks being performed and the number of available processors.

Here is an overview of the key components and steps described:

1. Thread Pool Management:
- The server application utilizes a pool of execution threads, referred to as worker threads, for performing requested tasks.
- The thread pool manager handles incoming task requests asynchronously and queues them for available worker threads.

2. Optimizing Thread Pool Size:
- The optimum thread pool size depends on the type of tasks being performed by the processors.
- For I/O-related tasks, a larger number of worker threads is considered efficient to utilize processing bandwidth effectively.
- Computational tasks, which are processor-intensive, may benefit from limiting the number of worker threads to avoid wasteful context switching.

3. Dynamic Thread Pool Size Adjustment:
- The challenge in optimizing the thread pool is compounded by processor scalability concerns, especially in systems with multiple processors.

- Lock contention issues can arise, and the text discusses the negative scalability impact of using a large number of threads in certain situations.

4. Gating Function and Exit Function:

- The thread pool manager incorporates a gating function and an exit function to regulate the number of active threads dynamically.
- The gating function compares the current number of active threads against a thread limit. If the limit is reached, it may delay the scaling of a new thread until the count drops below the limit.

5. Adjusting Thread Limit Based on CPU Utilization:

- The thread limit is updated at predefined intervals, such as one second.
- An update function checks the current CPU utilization. If the utilization is below a defined lower threshold, the thread limit is increased; if it's above an upper threshold, the thread limit is decreased.

In summary, the technique aims to dynamically adjust the number of active threads in a thread pool based on the varying conditions of processor loads. It addresses challenges associated with processor scalability and seeks to optimize the performance of a server application by efficiently managing the execution threads.

4. DATASETS

Dynamic thread pool management techniques are pivotal in modern computing environments to efficiently handle varying workloads and system conditions. Evaluating the performance of these techniques involves utilizing diverse datasets and performance measures to assess their effectiveness and efficiency comprehensively. This section explores the datasets commonly used in evaluating dynamic thread pool management techniques, as well as the performance measures employed to quantify their performance in real-world scenarios.

1. *Synthetic Workloads:*

Synthetic workloads play a crucial role in evaluating dynamic thread pool management techniques by allowing researchers to simulate various workload scenarios under controlled conditions. These workloads are artificially generated with parameters such as task arrival rates, execution times, and dependencies to mimic real-world scenarios. For instance, researchers might simulate web server workloads with a mix of CPU-bound and I/O-bound tasks to evaluate the dynamic allocation of threads in response to different types of tasks. Synthetic workloads provide flexibility in exploring a wide range of workload characteristics and their impact on thread pool performance.

In the reference paper [2], they consider a synthetic workload designed to simulate a web server environment with varying traffic patterns, including peak hours with high request rates and off-peak hours with low request rates. By adjusting parameters such as request arrival rates and task execution times, researchers analyze the performance of dynamic thread pool management techniques under different workload intensities.

2. *Real-World Workloads:*

Real-world datasets provide valuable insights into the behavior of dynamic thread pool management techniques in actual production environments. These datasets are often collected from production systems or benchmark suites representing diverse computing environments such as web servers, databases, and scientific applications.

In the reference paper [6], they consider a real-world workload dataset collected from a server system consisting of Pentium III 800MHz CPU with 256MB RAM, and the user system consists of Pentium III 800MHz SMP CPU with 512MB RAM. In addition, a 100Mbps Switched Fast Ethernet network is also installed. The number of users per unit time is decided using the Poison distribution. Then, the calculated number of threads tries to connect to the server at the same time, and the connected threads exchange the packet with the size of 100 Kbytes. Researchers use this dataset to evaluate the performance of dynamic thread pool management techniques in optimizing database query processing and resource utilization, thereby improving overall system efficiency.

3. *Benchmark Suites:*

Benchmark suites designed for evaluating concurrency and parallelism mechanisms also serve as valuable datasets for assessing dynamic thread pool management techniques. These suites include standardized workloads and performance benchmarks representing a variety of computational tasks and workload characteristics. For instance, the PARSEC Benchmark Suite, SPLASH-2 Benchmark Suite, and SPEC CPU Benchmark Suite provide diverse workloads for evaluating thread pool performance across different computing paradigms. Researchers leverage benchmark suites to benchmark their techniques against established standards and compare their performance against state-of-the-art approaches in the field.

In the reference paper [7], they consider a set of benchmark suites and run them for various resource manager (RM) configurations. During the benchmark, the Managed System provides the RM with performance metrics such as throughputs, and the RM returns resource allocations such as concurrency levels. This approach scales poorly. Typically, it takes minutes or hours to run a performance benchmark, and the number of configurations to evaluate is quite large.

4. *Trace-Based Workloads:*

Trace-based workloads are derived from system execution traces captured during the operation of real applications or systems. These traces provide detailed information about the sequence of operations, resource utilization, and inter-thread dependencies, offering researchers insights into the behavior of dynamic thread pool management techniques under real-world conditions. For example, researchers might analyze execution traces from distributed systems to understand task migration patterns and resource contention scenarios, enabling them to optimize thread pool management strategies accordingly. Trace-based workloads facilitate the evaluation of thread pool performance with empirical data, enhancing the credibility and relevance of research findings.

In the reference paper [8], the authors consider a trace-based workload derived from an authentication server of traffic addressed to Wikipedia. Sampling from the original trace of Wikipedia is used to create the traces of various request rates, which ensures realistic performance evaluation. Here the password of the user account is encoded with MD5 algorithm. The client and servers have AMD Dual Core 2.80 GHz processor, 2GB main memory, Window 7 OS, and Sun JDK 6. Researchers use this workload to analyze the performance of dynamic thread pool management techniques in optimizing task allocation and load balancing across distributed computing nodes, thereby improving overall system scalability and efficiency.

5. PERFORMANCE METRICS AND COMPARISON

1. *Throughput:*

Throughput is a fundamental performance metric for evaluating dynamic thread pool management techniques as it measures the rate at which tasks are completed by the system over a specific period. In the context of thread pool management, throughput quantifies the system's capacity to process incoming tasks efficiently and maximize resource utilization. Researchers measure throughput in terms of tasks completed per unit of time, such as tasks per second or requests per minute. Higher throughput indicates better system performance and scalability, demonstrating the effectiveness of dynamic thread pool management techniques in handling varying workloads and workload spikes. Researchers measure throughput by counting the number of tasks completed by the system per second under varying workload conditions. Higher throughput values indicate improved system performance and efficiency, demonstrating the effectiveness of the dynamic thread pool management technique in maximizing resource utilization and task processing capacity.

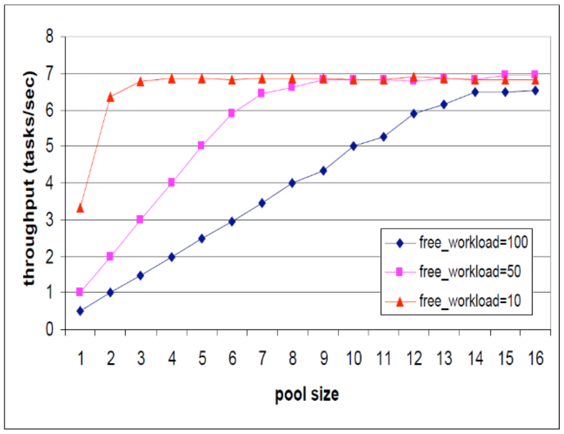

Fig 3: Throughput Vs. pool size

Fig 3 shows the relationship between the throughput and the thread pool size. First, we choose free_workload = 100. From this figure, it is obvious that the throughput of multithreaded applications can be improved proportional to the pool size when the pool size is relatively small. Unfortunately, such improvement cannot be sustained when the pool size is greater than the threshold. They suspect this phenomenon is caused by two issues. First, the application can only benefit from using a limited number of threads. When the pool size passes this threshold, the capacity of the application to utilize available threads becomes saturated and no performance improvement can be obtained. Second, the maintenance overhead brought by increasing pool size might overshadow the benefits obtained by using more threads.

The author Dongping Xu [5] provides a tangent to this performance metric, and calls it the Reciprocal of Average Idle Time (RAIT) which is defined as $RAIT_i$

= AIT_1/AIT_i, where AIT_1 is the average idle time of pool size 1, and AIT_i is the average idle time of pool size i.

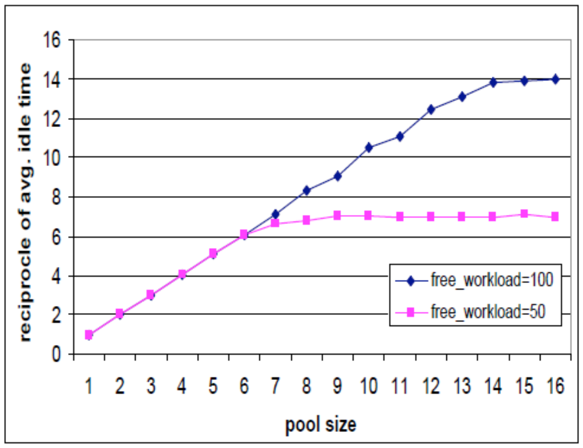

Fig 4: RAIT Vs. pool size

The author suggests that by comparing Fig 3 with Fig 4, the average Idle time of tasks has a strong correlation to system throughput. These results indicate the possibility of using AIT to infer the potential throughput of multithreaded programs. We can use such information to adjust the thread pool size on the fly.

2. *Response Time:*

Response time is a critical performance metric that quantifies the time interval between task submission and task completion. In the context of dynamic thread pool management, response time measures the system's responsiveness to incoming tasks and reflects the efficiency of task scheduling and execution. Researchers evaluate response time to assess the latency experienced by tasks in the system and optimize thread pool management strategies to minimize response time and enhance user experience. Response time is typically measured in milliseconds or microseconds, providing insights into the system's ability to meet performance targets and deliver timely responses to user requests.

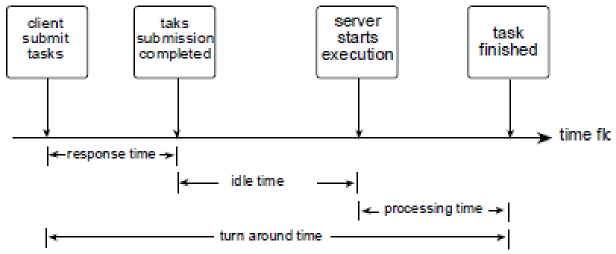

Fig 5 : The time flow of a submitted task

The author Sheraz Ahmed [2] considers a dynamic thread pool management technique implemented in a web server environment. Researchers measure response time by recording the time taken for each incoming HTTP request to be processed and responded to by the server. Lower response time values indicate improved system responsiveness and user experience, demonstrating the effectiveness of the dynamic thread pool management technique in minimizing task latency and optimizing task execution.

3. *Synthetic Workloads:*

Pool size adjustment is a performance measure that evaluates the dynamic allocation of threads in the thread pool based on workload characteristics and system conditions. Dynamic thread pool management techniques adjust the number of available threads dynamically to optimize resource utilization and system performance. Researchers analyze pool size adjustment mechanisms to assess their effectiveness in adapting to changing workload patterns and workload intensities. Pool size adjustment strategies aim to strike a balance between maximizing throughput and minimizing resource contention, ensuring optimal performance under varying workload conditions.

The author Dongping Xu [5] considers a dynamic thread pool management technique implemented in a distributed computing environment with fluctuating workload patterns. He examined the behavior of the dynamic thread adjustment algorithm when it is used in real applications. To do that, he implemented the dynamicThreadPool algorithm in their thread pool system. This algorithm is executed at the end of each cycle, which is defined as five completed jobs in their experiments. For thread pools with different initial pool sizes, the behavior of this algorithm might be different. Therefore, they have chosen two initial thread pool sizes, 4 and 16, for the experiments. The results are shown in Fig 6. The experimental results show that, for both initial thread pool sizes, the algorithm continuously increases the thread pool size towards the safe zone. The pool size becomes stable around this area. Note that the maximal adjusted pool size is not constant when it reaches the stable area. Instead, it fluctuates around some fixed value. In a perfect environment, this size would be the same. However, on real machines, the AIT we obtain might vary due to other factors (such as OS workloads and job behaviors). This will affect the accuracy of AIT and our algorithm. Therefore, some fluctuation is acceptable for our algorithm.

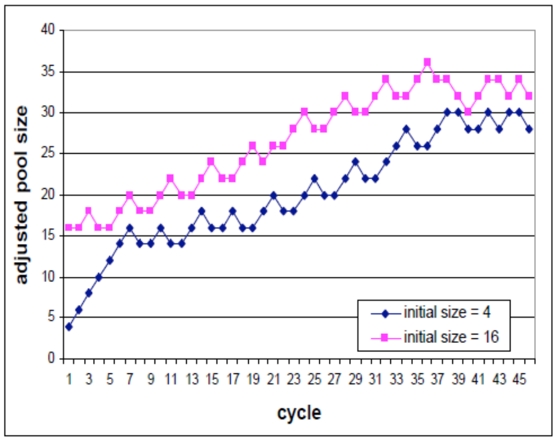

Fig 6. The adjusted thread pool size for two different initial thread pool sizes.

In the second experiment the author compares the throughputs of the original thread pool (which is called the static thread pool) and the dynamic thread pool. The dynamic thread pool is designed to adjust the pool size according to the behavior of multithreaded applications. The goal is to achieve better performance without introducing too much overhead. Therefore, comparing throughput will help to understand the performance improvement brought by using the dynamic thread pool. The experimental results are shown in Fig 7. To compare the performance more clearly, the throughput of the dynamic thread pool is normalized to the throughput of the static thread pool.

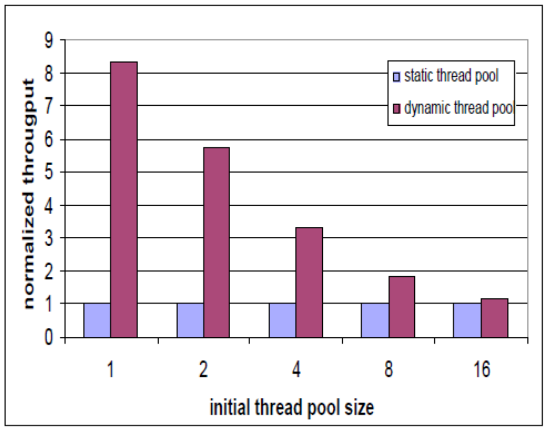

Fig 7. The throughput improvement by using dynamic thread pool.

The experimental results clearly show the performance improvement brought about by using a dynamic thread pool. For a thread pool with initial thread number =1, the throughput of the dynamic thread pool is about 8 times that of the static one. For other initial pool sizes, similar improvements are also observed. Interestingly, the improvement drops gradually when the initial thread number increases. This is because the performance of a static thread pool is already closer to optimal when the initial pool size is large.

4. *Resource Utilization:*

Resource utilization metrics such as CPU usage, memory consumption, and thread utilization provide insights into how effectively system resources are utilized by dynamic thread pool management techniques. Optimizing resource utilization is crucial for maximizing system performance and scalability while minimizing operational costs. Researchers analyze resource utilization metrics to identify bottlenecks and optimize thread pool management strategies to improve overall system efficiency. By monitoring resource utilization patterns, researchers can fine-tune thread pool parameters and scheduling policies to achieve optimal resource allocation and maximize system throughput.

In the reference paper [8] by Kang-Lyul lee, the authors consider a self-adaptive dynamic thread pool management technique called trendy exponential moving average (TEMA) for predicting and retaining a proper number of threads in the pool, which minimizes the response time and maximizes the resource utilization. Researchers measure resource utilization metrics such as CPU usage and memory consumption to assess the effectiveness of the thread pool management technique in optimizing resource allocation and minimizing resource wastage. Lower resource utilization values indicate improved system efficiency and cost-effectiveness, demonstrating the effectiveness of the dynamic thread pool management technique in maximizing resource utilization.

The Author Fabrizio Muscarella [9] , in his patent, also uses the method and apparatus that enable adaptive thread management, which is made to be dynamic based on system resource utilization. He proposes a thread pool adjuster that changes the number of threads in the thread pool based on the current resource consumption and a prediction of thread performance, the prediction based on the cumulative thread performance data, including adding a thread to the bottom of the thread pool to expand the thread pool, or closing threads that are inactive for the threshold period of time, the threshold period of time indicated by the cumulative thread performance data; and wherein the threads are returned to the top of the thread pool when execution of the one or more tasks by the threads are completed.

5. Scalability:

Scalability is a performance measure that evaluates the system's ability to handle increasing workload sizes and system loads while maintaining performance. Dynamic thread pool management techniques aim to scale seamlessly with growing workload demands by dynamically adjusting thread pool parameters and workload distribution strategies. Researchers assess scalability by measuring throughput and response time under varying workload conditions and system configurations. Scalability testing helps identify performance bottlenecks and scalability limitations, enabling researchers to optimize thread pool management techniques for enhanced scalability and performance.

In Robsman's [4] patent, the author considers a dynamic thread pool management technique implemented in a distributed computing environment with increasing workload demands. His research evaluates scalability by measuring system throughput and response time as the workload intensity increases. Effective scalability ensures that the system can handle growing workload sizes without significant degradation in performance, demonstrating the effectiveness of the dynamic thread pool management technique in maximizing system scalability and efficiency.

7. Fairness:

Fairness is an essential performance measure that evaluates the equitable distribution of resources among concurrent tasks or threads in the system. Dynamic thread pool management techniques should ensure fair allocation of resources to prevent starvation or resource contention, thereby improving overall system performance and user satisfaction. Researchers evaluate fairness by analyzing task scheduling algorithms and resource allocation strategies to ensure that all tasks receive fair treatment and timely processing. Considering dynamic thread pool management techniques deployed in a multi-user server environment, researchers evaluate fairness by analyzing task scheduling algorithms to ensure that all users receive fair access to system resources and timely processing of their tasks. Fairness metrics such as task response times and resource allocation fairness are used to assess the effectiveness of the thread pool management technique in ensuring fair resource allocation and user satisfaction.

8. Overhead:

Overhead refers to the additional computational or operational costs incurred by dynamic thread pool management techniques in managing thread pools and scheduling tasks. Minimizing overhead is essential for maximizing system performance and efficiency while minimizing resource consumption and operational costs. Researchers analyze overhead metrics to assess the computational and memory overhead associated with thread pool management techniques and identify opportunities for optimization. By minimizing overhead, researchers can improve system scalability and performance, enabling more efficient utilization of system resources.

In Dongping Xu's [5] paper, he mentions that the most attractive benefit of using a thread pool is to avoid the overhead of thread creation. However, that does not mean users should create a thread pool as large as possible. Indeed, the overhead for managing threads in the pool can be a big issue. To examine this problem, they studied the thread pool management overhead in their experiments. The most straightforward way of studying the management overhead is to increase the pool size. According to the previous discussion, the performance improvements brought by a thread pool will be saturated when the pool size reaches a threshold. After that, increasing the pool size will not help to improve the performance further. Instead, the overhead of thread pool management will degrade the performance (throughput) when the size becomes larger. Therefore, by increasing the pool size, we should be able to observe the impact brought by the overhead. Following this idea, we have measured the throughput of our benchmark by increasing the pool size from 1 to 55. The results are presented in Figure 7. According to this figure, the throughput becomes stable when the pool size reaches 13. After that the throughput mostly fluctuates around a fixed value. (The best throughput is observed when the pool size reaches 38. After that, it begins to drop gradually.) as shown in Fig 8.

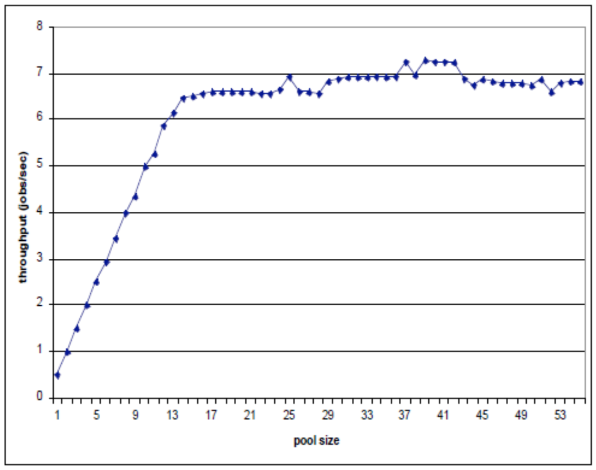

Fig 8: Thread Pool Management Overhead (Free_workload=100)

Consider a dynamic thread pool management technique implemented in a resource-constrained embedded system. Researchers measure overhead metrics such as thread creation overhead and synchronization overhead to assess the computational costs associated with the thread pool management technique. Lower overhead values indicate reduced computational costs and improved system efficiency, demonstrating the effectiveness of the dynamic thread pool management technique in minimizing resource consumption and operational overhead.

8. *Adaptability:*

Adaptability is a performance measure that evaluates the system's ability to adapt to changing workload conditions and system dynamics dynamically. Dynamic thread pool management techniques adjust thread pool parameters and scheduling policies in response to varying workload patterns and system conditions to optimize performance and resource utilization. Researchers assess adaptability by measuring how well thread pool management techniques respond to workload fluctuations and adapt to changing system loads. Adaptability testing helps identify opportunities for improving thread pool management techniques to achieve optimal performance and scalability under dynamic operating conditions.

The author Dongping Xu, in the reference paper [5] considers a dynamic thread pool management technique deployed in a cloud computing environment with fluctuating workload demands. The Researcher evaluates adaptability by measuring the system's response to workload fluctuations and its ability to dynamically adjust thread pool parameters such as thread allocation and task scheduling policies. Effective adaptability ensures optimal performance and resource utilization under varying workload conditions, maximizing system efficiency and responsiveness.

9. *Quality of Service (QoS):*

Quality of Service (QoS) is a performance measure that evaluates the system's ability to meet the requirements and expectations of submitted tasks in terms of responsiveness and reliability. Dynamic thread pool management techniques aim to ensure prompt response and fair treatment for all submitted tasks to meet quality of service objectives. Researchers analyze QoS metrics to assess the system's ability to deliver timely responses and meet performance targets under varying workload conditions. QoS testing helps identify opportunities for improving thread pool management techniques to achieve optimal responsiveness and reliability in task processing.

The author Dongping Xu [5] uses a dynamic thread pool management technique deployed in a real-time processing environment with strict performance requirements, and mainly focuses on the QoS in meeting the requests of submitted tasks, thread pool and the operating system. He evaluates QoS metrics such as task response times and reliability of task execution to assess the system's ability to meet performance targets and ensure timely processing of critical tasks. High QoS values indicate improved system reliability and responsiveness, demonstrating the effectiveness of the dynamic thread pool management technique in meeting performance requirements.

Evaluating the performance of dynamic thread pool management techniques involves utilizing diverse datasets and performance measures to assess their effectiveness and efficiency comprehensively. By leveraging these datasets researchers can simulate various workload scenarios and evaluate thread pool performance under realistic conditions. Additionally, by measuring performance using these metrics, researchers can systematically compare different thread pool management techniques and identify opportunities for optimization. By conducting experiments using representative datasets and performance measures, researchers can gain valuable insights into the behavior and performance of dynamic thread pool management techniques, ultimately improving system efficiency, scalability, and user satisfaction in modern computing environments.

6. DESIGN CONSIDERATIONS

Design and implementation of dynamic thread pool management techniques encompass a multitude of critical considerations and challenges that significantly influence system performance, scalability, and resource utilization. As foundational components of concurrent and parallel computing environments, these techniques play a major role in orchestrating the execution of tasks across multiple threads, thereby maximizing system throughput and responsiveness. However, achieving optimal performance requires addressing various design and implementation issues that arise due to the inherently complex nature of dynamic thread pool management.

One of the foremost challenges lies in ensuring concurrency control and synchronization mechanisms, where preventing race conditions and ensuring thread safety are paramount. This necessitates the careful use of synchronization primitives such as locks, semaphores, and mutexes to coordinate access to shared resources and maintain data consistency across threads. However, overuse of synchronization mechanisms can introduce contention and reduce concurrency, ultimately impacting system performance. Therefore, striking a balance between ensuring thread safety and minimizing synchronization overhead is crucial for effective dynamic thread pool management.

Efficient task partitioning and distribution among threads are also fundamental design considerations, ensuring load balancing across threads and optimizing resource utilization. Various strategies, such as work-stealing or task queues, are employed to achieve efficient task distribution. Work-stealing allows idle threads to steal tasks from other threads' queues, thereby ensuring workload balance and preventing individual threads from becoming overloaded or underutilized. However, designing efficient task partitioning algorithms and load balancing policies that minimize contention and maximize throughput remains a challenge in dynamic thread pool management.

Dynamic resource management is another critical aspect of designing dynamic thread pool management techniques. The ability to dynamically adjust the size and configuration of the thread pool based on workload demands is essential for optimizing resource utilization and responsiveness. Real-time monitoring of system parameters and workload metrics is necessary to make informed decisions about resource allocation and resizing. Efficient data structures and algorithms are needed to manage the dynamic resizing of the thread pool while minimizing overhead and ensuring responsiveness. However, achieving efficient and scalable dynamic resource management poses significant challenges, particularly in highly dynamic and unpredictable environments.

Moreover, managing overhead and minimizing its impact on system performance is a key consideration in the design and implementation of dynamic thread pool management techniques. Dynamic resizing, workload prediction, and load balancing algorithms introduce overhead in the form of additional computational and memory resources required for monitoring, resizing, and task distribution. Minimizing overhead while maximizing system performance is crucial for achieving optimal performance in dynamic thread pool management. Techniques such as adaptive resizing, workload prediction, and efficient task distribution algorithms aim to optimize performance while minimizing overhead. However, striking a balance between performance optimization and overhead reduction remains a challenging aspect of dynamic thread pool management.

Scalability and load balancing are also fundamental concerns, Ensuring scalability and efficient load balancing are essential for handling varying workload intensities and maximizing resource utilization. Strategies such as partitioning the workload or dynamically adjusting the size of thread pools based on workload metrics help maintain scalability and load balance. However, achieving efficient scalability and load balancing across varying workload conditions remains a complex and challenging aspect of dynamic thread pool management.

Furthermore, fault tolerance, error handling, and support for heterogeneous environments are critical aspects, ensuring system reliability, availability, and efficient utilization of diverse hardware resources. Designing robust error handling mechanisms and fault tolerance strategies is essential to ensure system reliability and availability. Techniques such as task reassignment, fault recovery, and graceful degradation help mitigate the impact of failures and errors in dynamic thread pool management. However, designing and implementing effective fault tolerance and error handling mechanisms present significant challenges, particularly in highly dynamic and unpredictable environments.

Another crucial aspect of designing dynamic thread pool management techniques is understanding workload characteristics and predicting future workload patterns which help anticipate changes in workload intensity and optimize resource allocation accordingly. Techniques for workload prediction aim to forecast future workload patterns based on historical data or predictive models, enabling proactive adjustment of thread pool resources and optimization of performance. However, achieving accurate workload prediction and proactive resource allocation remains a challenging aspect of dynamic thread pool management.

Moreover, performance monitoring and analysis are essential for evaluating the effectiveness of dynamic thread pool management techniques. Real-time performance monitoring and analysis provide valuable insights into system behavior and performance bottlenecks. Metrics such as throughput, latency, scalability, and resource utilization are used to assess system performance and identify potential areas for optimization. However, designing effective performance monitoring and analysis mechanisms that provide actionable insights into system behavior remains a challenging aspect of dynamic thread pool management.

Addressing these challenges is crucial for achieving optimal performance and resource utilization in dynamic thread pool management. adopting appropriate design principles and techniques, developers can design and implement robust and efficient dynamic thread pool management techniques that effectively orchestrate the execution of tasks in concurrent and parallel computing environments, ensuring optimal performance and resource utilization.

7. RECOMMENDATIONS

Deciding when to implement a particular dynamic thread pool management technique depends on various factors, including the characteristics of the application, the expected workload patterns, performance requirements, and system constraints. By carefully considering these factors, developers can make informed decisions about which technique best suits their specific use case.

For applications with predictable and stable workloads, fixed-size thread pools are suitable. These pools are straightforward to implement and understand, making them ideal for scenarios prioritizing simplicity and predictability over scalability and adaptability. Consider using fixed-size thread pools in applications such as batch processing systems, where the workload remains consistent and can be efficiently processed by a fixed number of threads.

In contrast, cached thread pools are well-suited for applications with fluctuating and unpredictable workloads. These pools dynamically adjust their size based on workload demands, optimizing resource utilization and responsiveness. Employ cached thread pools in applications such as web servers, where the workload can vary significantly based on user traffic and request patterns. They efficiently scale up or down to handle fluctuations in demand while minimizing resource overhead during periods of low activity.

Work-stealing thread pools are beneficial for applications with highly parallelizable workloads, excelling in load balancing and maximizing resource utilization. Integrate work-stealing thread pools in applications such as scientific simulations, where tasks can be divided into smaller units processed independently by multiple threads. These pools effectively scale with increasing workload intensity, dynamically adapting to changes in demand.

Adaptive thread pools are ideal for applications with dynamic and unpredictable workloads, continuously monitoring system parameters and workload metrics to optimize resource utilization and responsiveness. Utilize adaptive thread pools in applications such as real-time data processing systems or online gaming platforms, where workload patterns vary significantly over time. They proactively scale resources based on workload predictions, minimizing response times and improving throughput under varying workload intensities.

Hybrid approaches combining multiple thread pool management techniques offer a flexible solution to address specific requirements effectively. By leveraging the strengths of different techniques, hybrid approaches achieve optimal performance and resource utilization. Deploy hybrid approaches in applications with complex workload patterns or specific performance goals. For example, a combination of cached and adaptive thread pools may suit applications with both predictable and unpredictable workload components, ensuring efficient resource utilization and scalability across varying workload conditions.

The decision of when to employ a particular dynamic thread pool management technique hinge upon the precise demands of the application, encompassing workload nuances, performance objectives, and system limitations. Implementing such techniques mandates a nuanced comprehension of the distinct features and requisites of diverse applications. Accordingly, the selection of a specific technique should harmonize with the inherent nature of the application and its workload.

For real-time systems with stringent performance requirements, the analysis of optimal thread pool size becomes relevant, along with the integration of predictive schemes to anticipate workload changes effectively. In the emerging landscape of IoT and edge computing, where resource constraints and distributed architectures prevail, self-adaptive mechanisms and prediction-based schemes offer valuable insights for efficient resource allocation. High-performance computing applications benefit from predictive and self-adaptive techniques, ensuring optimal resource utilization and load balancing. In web servers and cloud computing environments, prediction-based schemes enhance responsiveness and scalability amidst dynamic workload variations.

These recommendations provide a guideline for selecting dynamic thread pool management techniques based on the specific requirements and characteristics of different application domains. It's essential to consider the unique features and challenges of each application to tailor the chosen technique for optimal performance and resource utilization.

8. FUTURE IMPROVEMENTS

A compelling avenue for improvement in dynamic thread pool management lies in the development of a hybrid model that synthesizes key elements from existing approaches. Combining insights from a variety of methodologies, the hybrid approach is designed to create a unified framework that offers enhanced versatility and efficiency. An adaptive mechanism is integrated into the hybrid model to enable dynamic thread pool adjustments according to workload. This adaptability ensures optimal resource allocation and task scheduling, thereby improving overall system efficiency. To further enhance performance, load balancing principles are incorporated to ensure the equitable distribution of tasks across processing units.

Moreover, the hybrid model incorporates an analysis of optimal thread pool size, allowing for tailored sizing based on specific workload characteristics. This optimization strategy enhances resource utilization and responsiveness to application demands, contributing to improved overall system performance. Furthermore, predictive capabilities are integrated into the hybrid model to accurately anticipate changes in workload patterns. By leveraging predictive modeling techniques, the hybrid approach can proactively adapt to evolving workloads, thereby enhancing system scalability and responsiveness.

The practicality of this hybrid model comes from its comprehensive approach, which addresses the diverse requirements of real-world applications. This hybrid model offers a versatile solution suitable for a variety of scenarios by combining adaptive mechanisms, load balancing principles, optimal sizing considerations, and predictive capabilities. By leveraging the strengths of existing techniques, this approach ensures adaptability, efficiency, and scalability across various application domains, thereby paving the way for enhanced performance and resource utilization.

9. CONCLUSION

In conclusion, dynamic thread pool management techniques play a pivotal role in optimizing system performance and resource utilization in concurrent and parallel computing environments. Throughout this paper, we have explored various dynamic thread pool management techniques, including fixed-size thread pools, cached thread pools, work-stealing thread pools, adaptive thread pools, and hybrid approaches. Each technique presents its own set of strengths and weaknesses, design considerations, and future scope for improvement.

Fixed-size thread pools offer simplicity and predictability, making them suitable for applications with stable workloads. However, their rigid nature limits scalability and adaptability to fluctuating workloads, leading to potential resource underutilization and performance bottlenecks under heavy loads. Cached thread pools dynamically adjust their size based on workload demands, optimizing resource utilization and responsiveness. While they offer flexibility and scalability, the overhead associated with dynamic resizing and load balancing algorithms can impact system performance and stability. Work-stealing thread pools excel in load balancing and scalability by distributing tasks efficiently among threads. However, the complexity of implementing efficient task-stealing algorithms and the risk of thread starvation pose challenges, particularly in highly parallelizable workloads. Adaptive thread pools continuously monitor system parameters and workload metrics to dynamically adjust pool size and configuration. Despite their effectiveness in handling dynamic workloads, the complexity of workload prediction and continuous monitoring introduces overhead and requires accurate predictions for optimal performance. Hybrid approaches combine multiple thread pool management techniques to leverage their respective strengths and address specific requirements more effectively. While hybrid approaches offer flexibility and customization, integrating different techniques increases implementation complexity and requires careful trade-off considerations.

Design and implementation issues such as concurrency control, task partitioning and distribution, dynamic resource management, overhead management, scalability, fault tolerance, workload prediction, and performance monitoring and analysis are critical considerations in dynamic thread pool management. Addressing these issues is essential for achieving optimal performance and resource utilization in dynamic computing environments. Looking ahead, future improvements in dynamic thread pool management techniques hold promise for addressing existing challenges and enhancing performance. Advancements in workload prediction, load balancing strategies, resource management techniques, autonomic thread pool management, integration with container orchestration systems, hybridization of techniques, performance monitoring and analysis, and support for heterogeneous architectures are areas of future scope for improvement.

While each dynamic thread pool management technique has its own strengths and weaknesses, if we were to pick a winner that is best suitable for most applications based on a balance of factors, the cached thread pool technique stands out as a versatile and effective choice. Its dynamic resizing capability, efficiency, and simplicity make it well-suited for handling varying workload intensities and optimizing resource utilization in concurrent and parallel computing environments. Dynamic thread pool management techniques are essential components of modern concurrent and parallel computing environments. By carefully considering the strengths and weaknesses of each technique, along with design considerations and future scope for improvement, developers can make informed decisions to choose the most appropriate dynamic thread pool management technique for their specific use case, thereby optimizing system performance and resource utilization.

10. ACKNOWLEDGEMENTS

We extend our deepest gratitude to Professor Yuchen Huang and Adjunct Support Faculty Venkatesh Srinivas for their invaluable contributions to our team in completing this survey paper. Their expertise, guidance, and unwavering support have been instrumental in steering our survey towards success. We are immensely grateful for their mentorship, encouragement, and insights, which have greatly enriched the depth and quality of our paper.

11. REFERENCES

[1] The Managed Thread Pool https://msdn.microsoft.com/enus/library/0ka9477y(v=vs.110).aspx (accessed on 10 July 2018).

[2] S. Ahmad, F. Bahadur, F. Kanwal, R. Shah, “Load balancing in distributed framework for frequency based thread pools,” Computational Ecology and Software, Vol. 6, No. 4, pp. 150-164, 2016.

[3] Y. Ling , T. Mullen , X. Lin, “Analysis of optimal thread pool size,” ACM SIGOPS Operating Systems Review, Vol. 34, No. 2, pp.42-55, 2000.

[4] D. Robsman, “Thread optimization,” US6477561 B1, 2002.

[5] D. Xu , B. Bode, “Performance Study and Dynamic Optimization Design for Thread Pool System,” In Proc. of the Int. Conf. on Computing Communications and Control Technologies, pp.167-174, 2004.

[6] D. Kang , S. Han , S. Yoo , S. Park, “Prediction based Dynamic Thread Pool Scheme for Efficient Resource Usage,” In Proc. of the IEEE 8th Int. Conf. on Computer and Information Technology Workshop, IEEE Computer Society, pp. 159-164, 2008.

[7] J. Hellerstein, “Configuring resource managers using model fuzzing: A case study of the .NET thread pool,” IFIP/IEEE International Symposium on Integrated Network Management (IM '09), pp. 1-8, 2009.

[8] K. Lee , H. Pham , H. Kim , H. Youn , O. Song, “A novel predictive and self-adaptive dynamic thread pool management,” In: Proceedings - 9th IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 93–98, 2011.

[9] Faisal Bahadur, Arif Iqbal Umar, Fahad Khurshid, “Dynamic Tuning and Overload Management of Thread Pool System”, In: International Journal of Advanced Computer Science and Applications, Vol. 9, No. 11, 2018

[10] R.T. Gowda, “Dynamic thread pool management,” US 8381216 B2, 2013.