Investigating Dynamic vs Static Thread Pools for performance in Multithreaded Applications

HARSHIL PATEL, University of California, Davis, USA
ANUGYA SHARMA, University of California, Davis, USA
KAICHI XIE, University of California, Davis, USA
ALFREDO ORTIZ, University of California, Davis, USA

1 Introduction & Motivation

The advancements in multicore architecture and multiprocessor computers are increasing and ongoing [2]. To keep up with this trend, we need techniques like multithreading to maximize the performance of ever-advancing cpu's [1]. Multithreading is of vital importance for this because it can harness the power of multiprocessor computer, improve system reliability by preventing one operation from negatively affecting another in a program(user interface event affecting time-critical operation), and maximize cpu use (in programs that read/write from a file, perform I/O, or poll the user interface) [1].

While a very efficient tool for harnessing the power of multiprocessor computer, multithreading does come with some challenges. The top most important challenge in multithreading is the management of thread pools [7], namely the size of the thread pool. The number of threads in a thread pool determine the changes in response times and resource utilization [7]. In our research, we aim to collect empirical data that informs decisions on thread pool size. In doing so, we also aim to propose a "dynamic thread pool" with varying number of threads such that we can optimize minimum response time for maximum resource utilization. We will evaluate evaluate the performance trade-offs between static and dynamic thread pools under varying workloads and system configurations and answer the question: At what point does the overhead of dynamic thread creation outweigh its memory savings compared to static thread pools?

2 Background & Related Work

Current heuristics that are used to determine the number of threads are flawed [8]. In our preliminary research, we find that there have been few researches on dynamic model for thread size. One of the first such model was the Watermark model which adjusts the number of threads according to the current number of incoming requests [6]. The issue with this model is that while it solves the problem of "efficient usage of resources", it doesn't efficiently obtain the required number of threads in advance [5]. Furthermore, there are also prediction based models to determine thread pool size, but in such models, only factors such as the request rate or number of worker threads is considered in the management [7]. One of the research that has proposed dynamic thread pool model(prediction based), themselves point out that the model lacks testing for a longer period of time and on general server environment which provides various services simultaneously [8]. Another research adds to the prediction model by proposing what they call the "Trendy exponential moving average model" and again the issue with this model seems to be that it doesn't have the desired accuracy in its prediction [7].

Of the many researches into dynamic sizes of pool, none look at the memory usage/CPU overheads and whether those are significantly lower than static pool to signify the performance cost. The trade off for replacing static threads affect the performance and costs of cloud-based systems [4], use of dynamic pools in web services handling could have latency spikes during high traffic. Not just costs and such latency issues, there is also the question of system design

such as database connection pools, task scheduling in distributed systems, etc which all use thread pools to manage concurrency. There is a lack of comprehensive studies that quantify the performance implications of static vs dynamic thread pools across different workloads and system configurations. By systematically evaluating these trade-offs, we can provide practical guidelines for system architects to choose the right thread pool strategy based on their specific workload and performance requirements.

3 Problem challenges

When implementing two different thread pools to compare their performance, one challenge is to make sure that the comparison is fair. We need to make sure that both implementations are the same in terms of how they execute instructions and the only difference is how they manage the threads. As shown by Desrochers [3], that the throughput gap between a lock-free queue and lock_based queues can range from 14x to 200x depending on the platform. We need to make sure that both implementations use the same data structures such as queues, locks, etc and the same algorithms to manage to schedule the tasks and threads.

4 Proposed Solution

We address the shortcomings mentioned in 2 by implementing minimal static and dynamic thread pool architectures in C++ that differ only in their thread lifecycle management policies, while sharing the same task scheduling and execution logic. The static thread pool will pre-create a fixed number of threads at initialization, which will remain active throughout the lifetime of the pool. The dynamic thread pool will start with zero threads and will create new threads on-demand when tasks arrive. Idle threads will be terminated after a configurable timeout period to free up resources. Importantly, both implementations will use the same task queue, task scheduling algorithm (e.g., FIFO), and other shared components to ensure a fair comparison. This approach will allow us to isolate architectural differences from quality unlike comparing off the shelf implementations like Intel TBB or Boost, which may have optimizations that effect performance independently of thread pool policy.

4.1 Evaluation

To evaluate our proposed thread pool implementations, we will benchmark them under three traffic patterns, each defined by a task arrival process that reflects different traffic scenarios:

- Steady load: a constant arrival rate of X tasks/second for the full duration of the run.
- Burst Load: a burst of X tasks/second followed by a Y-second idle period, repeated in cycles.
- Ramping Load: alternating phases on X tasks/second and Y tasks/second in repeated cycles.

We will also vary the number rate of incoming tasks to simulate different levels of system load. To isolate how task characteristics interact with thread-pool policy, we will apply two controlled micro-benchmarks:

- Memory-based tasks: $N \times N$ matrix multiplications, where N is tunable to scale memory pressure.
- Latency-based tasks: a synthetic task that blocks for a configurable sleep duration T to emulate downstream I/O or service latency.

For each task type, we evaluate four benchmark intensities: Light, Medium, Heavy, and Mixed. For latency-based tasks, each request performs a blocking sleep with $T \in \{100\,\mu s, 1\,ms, 10\,ms\}$ to represent Light, Medium, and Heavy service times, respectively. For memory-based tasks, we use $N \times N$ matrix multiplication with $N \in \{128, 256, 512\}$ to

represent Light, Medium, and Heavy working-set sizes, respectively. The Mixed workload captures heterogeneity by combining light, medium, and heavy tasks in varied proportions (e.g., 50% light, 30% medium, 20% heavy).

We will evaluate the following experiment matrix (Table 1):

Dimension	Values
Traffic Patterns	Steady load, Burst Load, Ramping Load
Task Types	Memory-based, Latency-based
Task Intensities	Light, Medium, Heavy, Mixed
Pool Types	Static, Dynamic
Core Counts	4 cores, 16 cores

Table 1. Experiment configuration matrix.

For each task arrival pattern, we will measure throughput (tasks completed per second), latency (p50, p95, p99 time from task submission to completion), PSS Memory (Proportional Set Size attributable RAM used by the process), and thread count (active threads over time). We will fix the CPU resources available to the benchmark by restricting it to exactly 4 or 16 cores using Linux CPU affinity (e.g., tasket). For each configuration, we run the benchmark for 120 seconds and repeat the process three times. we will then aggregate the metrics across the three trials and report the mean and variance.

5 Expected Results

For out project, we expect to see that the dynamic thread pool with outperform the static thread pool in scenarios with variable workloads, such as bursty or ramping traffic patterns. This is because the dynamic pool can adapt to changing demand by creating or terminating threads as needed, leading to better resource utilization. In contrast, the static thread pool might perform well under steady load conditions where the fixed number of threads can handle the incoming tasks without significant queuing delays. We also expect the dynamic thread pool to incur some overhead due to thread creation and destruction, which may lead to higher latency in scenarios with frequent load changes. However, we expect that this overhead will be offset by the improved responsiveness and reduced queuing delays in high load and variable traffic scenarios. In terms of memory usage, we expect the dynamic thread pools to use less overall memory when compared to the static thread pools during low load periods, as idle threads in the dynamic thread pool can be terminated to free up resources.

6 Timeline

We have broken down our timeline into weeks with six weeks to finish and present the project (Table 2).

Week	Tasks
Week 1 (2026-2-3)	Research dynamic and static thread pools, experiment with code to understand implementation
Week 2 (2026-2-8)	Start implementing the thread pools, traffic generation scripts and evaluation scripts, continue research as needed
Week 3 (2026-2-15)	Continue working on code, verify progress aligns with plan, fix bugs
Week 4 (2026-2-22)	Complete implementation and start evaluation
Week 5 (2026-3-1)	Complete results, write research paper, create presentation slides and practice
Week 6 (2026-3-8)	Finalize results, paper, and slides. Submit project and present

Table 2. Project timeline.

References

[1] [n. d.]. https://www.ni.com/docs/en-US/bundle/labview/page/benefits-of-multithreaded-applications.html?srsltid=AfmBOorcBduf0oUl2o4EN4wKzFmLMkZnnVQo8f_3tlVAGO8ZVuUntAoo

[2] George K Adam. 2022. Co-design of multicore hardware and multithreaded software for thread performance assessment on an FPGA. *Computers* 11, 5 (2022), 76.

[3] Cameron Desrochers. 2014. A Fast General Purpose Lock-Free Queue for C++. https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++ Accessed: 2026-02-02.

[4] Daniela L Freire, Angela Mazzonetto, Rafael Z Frantz, Fabricia Roos-Frantz, Sandro Sawicki, and Vitor Basto-Fernandes. 2021. Performance evaluation of thread pool configurations in the run-time systems of integration platforms. *International Journal of Business Process Integration and Management* 10, 3-4 (2021), 318–329.

[5] DongHyun Kang, Saeyoung Han, SeoHee Yoo, and Sungyong Park. 2008. Prediction-based dynamic thread pool scheme for efficient resource usage. In *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*. IEEE, 159–164.

[6] Ji Hoon Kim, Seungwok Han, Hyun Ko, and Hee Yong Youn. 2007. Prediction-based dynamic thread pool management of agent platform for ubiquitous computing. In *International Conference on Ubiquitous Intelligence and Computing*. Springer, 1098–1107.

[7] Kang-Lyul Lee, Hong Nhat Pham, Hee-seong Kim, Hee Yong Youn, and Ohyoung Song. 2011. A Novel Predictive and Self–Adaptive Dynamic Thread Pool Management. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 93–98.

[8] Yibei Ling, Tracy Mullen, and Xiaola Lin. 2000. Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review* 34, 2 (2000), 42–55.