Ecs251 project proposal

Harshil Patel[1], **Anugya Sharma**[1], **Kaichi Xie**[1]
[1]Department of Computer Science, University of California, Davis

1 Introduction & Motivation

The advancements in multicore architecture and multiprocessor computers are increasing and ongoing (Adam, 2022). To keep up with this trend, we need techniques like multithreading to maximize the performance of ever-advancing cpu's (ni). 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) (ni).

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 (Lee et al., 2011), namely the size of the thread pool. The number of threads in a thread pool determine the changes in response times and resource utilization (Lee et al., 2011). 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 (Ling et al., 2000). 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 (Kim et al., 2007). 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 (Kang et al., 2008). 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 (Lee et al., 2011). 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 (Ling et al., 2000). 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 (Lee et al., 2011).

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 (Freire et al., 2021), 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 (Desrochers, 2014), 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 Evaluation

Our first step is to implement two thread-pool methods: a static and a dynamic thread pool with C++. To compare them fairly, we will benchmark under three representative workload patterns, each defined by a task arrival process that reflects common real-world traffic conditions:

- Steady load: a constant arrival rate of 1,000 tasks/second for the full duration of the run.
- Flash crowd: a burst of 5,000 tasks/second followed by a 10-second idle period, repeated in cycles.
- Daily ramp: alternating phases on 100 tasks/second and 2,000 tasks/second in repeated cycles.

Across these workloads, we will evaluate two resource dimensions that influence the trade-off between static and dynamic thread pools: latency (time cost) and memory (space cost). 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.

We evaluate four task categories for both benchmarks. For each benchmark, we run (1) Light, (2) Medium, and (3) Heavy tasks by increasing the corresponding tuning parameter values (T for latency-based tasks and N for memory-based tasks). Specifically, each request for the latency-based tasks performs a blocking sleep with $T \in \{100\,\mu s, 1\,ms, 10\,ms\}$, to represent the Light, Medium, and Heavy service time. 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. Finally, we include (4) a Mixed workload for each constraint to capture heterogeneity: 60% Light / 30% Medium / 10% Heavy, where each request samples its T or N from the setting according to this distribution.

Metric	Definition
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)
Thread Count	Active threads over time

Table 1: Metrics collected in each benchmark run.

For each benchmark run, we will record four metrics to evaluate the time and memory costs accordingly (Table 1).

We will run 12 experiment configurations in total: three workload patterns $\times$ 2 pool types (static and dynamic) $\times$ 2 core counts (4 cores and 16 cores). 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 then aggregate the metrics across the three trials and report the mean and variance.

References

[link].

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

Cameron Desrochers. 2014. A fast general purpose lock-free queue for c++. Accessed: 2026-02-02.

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):318–329.

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*, pages 159–164. IEEE.

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*, pages 1098–1107. Springer.

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*, pages 93–98. IEEE.

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