Threads-Pools and Coroutines: An OS-Level Comparative Study of Concurrency in C++

Raiyan Jahangir
UC Davis

Ahnaf Faisal
UC Davis

Jun Wu
UC Davis

1 Introduction

1.1 Problem and Motivation

Modern systems software increasingly depends on concurrency to exploit multicore parallelism and to overlap computation with I/O [12]. In practice, two families of designs dominate: (i) thread pools, where the kernel schedules many OS threads, and the application manages a queueing/scheduling policy for tasks [13], and (ii) coroutine-based async execution, where tasks are cooperatively scheduled in user space and typically resume at well-defined suspension points [2]. Although developers often treat coroutines as "lightweight threads" [11], the actual performance and scalability outcomes depend on OS-visible effects such as kernel scheduling decisions, context-switch frequency, blocking behavior, synchronization, and oversubscription, especially under mixed CPU/I/O workloads [4]. This makes it difficult to choose between thread pools and coroutines based on intuition alone.

This project, therefore, takes an operating-systems perspective. We will compare multiple thread-pool designs and a C++20 [16] coroutine executor using the same workloads and explain performance using OS-level metrics such as CPU utilization and context switches, rather than application-level metrics like latency or throughput. Specifically, the proposal commits to evaluating four thread-pool variants: global queues vs. per-thread queues with work stealing, and fixed-size vs. elastic resizing, alongside a coroutine-based execution model, across CPU-bound, I/O-bound, and mixed workloads.

1.2 Related Works and their Limitations

Prior work offers several alternative approaches for handling large-scale concurrency, but each leaves gaps relative to our goal of an OS-grounded, apples-to-apples comparison of thread-pool designs versus coroutines. In server-side I/O, architectural solutions such as Flash (AMPED) [14] and SEDA [19] show that event-driven/staged designs can outperform thread-centric servers depending on workload characteristics, but these studies primarily compare *application architectures* rather than isolating thread-pool design dimensions (global vs. per-worker queues, work stealing, fixed vs. elastic sizing) against coroutine executors under a shared benchmark suite and OS-level instrumentation. Systems such as Capriccio [18] and kernel/runtime interfaces like Scheduler Activations [1] demonstrate that user-level scheduling can scale and can mitigate blocking/preemption issues, yet they focus on specialized runtimes or OS abstractions rather than evaluating modern, widely-used concurrency mechanisms (C++ thread pools and C++20 stackless coroutines) in mixed CPU/I/O settings with comparable implementations and measurements.

For CPU-parallel workloads, work-stealing schedulers [5, 6, 17] provide strong theoretical and runtime-level evidence for efficient task scheduling, but they largely target structured task graphs and CPU-centric execution. They do not fully capture end-to-end behavior under I/O blocking, synchronization, and OS scheduling, nor do they directly compare with coroutine-based cooperative scheduling. Finally, coroutine-focused studies in high-performance systems [8, 9] show compelling gains in domain-specific engines, but results often depend on bespoke runtimes and access patterns and are not directly transferable to general thread-pool architectures or to OS-level diagnostic comparisons. Collectively, these works motivate multiple viable solutions but leave a practical gap. That is, a controlled evaluation of thread-pool design choices versus coroutine executors across CPU-bound, I/O-bound, and mixed workloads, coupled with OS-level metrics to explain the observed trade-offs.

1.3 Challenges

One challenge with this problem is making fair comparisons between concurrency models that operate differently under the hood. Thread pools use kernel-managed threads and preemptive scheduling, while coroutines use user-level cooperative scheduling on top of OS threads. Even small differences in implementation, runtime optimization, or tasks can influence the results. Also, performance differences across models are not always easy to see with high-level metrics alone, such as latency, and require measuring OS-level performance, such as CPU utilization and context switches. Another significant challenge is creating workloads that can detect differences among models without biasing any one model, which is also not easy. Finally, from a coding perspective, thread-based concurrency is difficult to debug because errors such as deadlocks and race conditions depend on subtle timing and scheduling behavior that is often nondeterministic [7]. As a result, bugs may disappear under debugging or testing, making them hard to reproduce and fix.

1.4 Proposed Solution

In this project, we propose a systematic, controlled comparison of four thread pool designs and a C++20 coroutine-based execution model [20]. We will use the same workloads and benchmarks setup to evaluate the performance for all approaches. The thread pools chosen will cover two design dimensions: thread count and queue organization. To be more specific, we will implement and evaluate a fixed-size thread pool with a global queue, a ForkJoin-style fixed-size pool [10] with per-thread queues and work stealing [5], an elastic (dynamic) pool [15] with a global queue, and an advanced elastic pool that includes dynamic resizing with per-thread queues and work stealing (Table 1). This setup allows us to study several thread pool design choices that affect performance. Unlike prior work on event-driven or staged server architectures, which focuses on end-to-end server designs, our project isolates specific concurrency and scheduling mechanisms, enabling a clearer comparison of thread-pool design dimensions across a unified benchmark suite and controlled experimental conditions.

In addition, we will implement a coroutine-based version using C++20 coroutines [3], in which tasks are scheduled cooperatively by a user-level executor rather than by the kernel. We will test these implementations using a range of applications, including CPU-bound workloads such as matrix multiplication and Fibonacci computation, I/O-bound workloads such as an HTTP server handling concurrent requests, and mixed workloads that combine computation with asynchronous I/O. By running the same workloads across all implementations, we can compare their performance using metrics such as throughput and latency, and then use OS-level performance data, such as CPU utilization and context switches, to explain what is happening. While prior work on user-level threading and scheduling mechanisms explores specialized runtimes or OS interfaces, our project directly compares modern thread pools and C++20 coroutines using comparable implementations and focuses on OS-level measurements. Overall, this approach lets us see the trade-offs between different thread pool designs and coroutines, and keeps the focus on operating system behavior rather than framework-specific details.

Table 1: Thread Pool Designs

Pool Type	Thread Count	Queue Design
Classic Fixed Pool	Fixed	Global Queue
Forkjoin-style Pool	Fixed	Per-thread + stealing
Elastic Pool	Dynamic	Global Queue
Advanced Elastic Pool	Dynamic	Per-thread + stealing

1.5 Anticipated Results

We expect to see trade-offs among different thread pool designs and the coroutine-based model, rather than one approach consistently outperforming the others. Fixed-size thread pools with global queues may experience threads competing for the same lock as load increases, whereas work-stealing pools are likely to scale better for CPU-bound workloads but incur stealing overhead. Elastic thread pools might handle unexpected workloads more effectively but introduce additional overhead from dynamic resizing, thread creation, and deletion. Coroutines are expected to perform well on I/O-bound workloads due to reduced context switching, but may be slower on CPU-bound tasks. Overall, we expect OS-level metrics such as CPU utilization and context switches to help explain these performance differences.

2 Project Timeline

- Week 1 (Feb 4 - 8):
  - Read related work and set up the development and benchmarking environment
- Week 2 (Feb 9 - 15):
  - Implement CPU-bound benchmarks
  - Implement a fixed-size thread pool with a global queue
  - Design CPU-bound, I/O-bound, and mixed workloads
- Week 3 (Feb 16 - 22):
  - Implement ForkJoin-style fixed-size pool
  - Implement mixed workload benchmark
- Week 4 (Feb 23 - Mar 1):
  - Implement an elastic pool and an advanced elastic pool
  - Implement an I/O-bound application (HTTP server)
- Week 5 (Mar 2 - 8):
  - Implement C++20 coroutine-based execution model

  - Run all workloads across all implementations and collect metrics

- Week 6 (Mar 9 - 18):
  - Wrap up any unfinished previous works
  - Analyze results, generate graphs, and write the final report

3 Evaluation Plan

Workloads

- CPU-bound: matrix multiplication, Fibonacci / compute loops
- I/O-bound: simple HTTP server handling concurrent requests
- Mixed: computation followed by asynchronous I/O and additional computation
- Vary the concurrency level and the number of worker threads for all workloads

Metrics

- Application-level
  - Throughput (tasks/sec or requests/sec)
  - Latency (p50, p95, p99)
- OS-level
  - CPU utilization
  - Context switches
  - CPU migrations
- Memory:
  - Peak and average memory usage (RSS)
  - Context switches
  - CPU migrations

Tools

- perf: CPU utilization, context switches, CPU migrations
- /usr/bin/time -v: runtime and memory usage
- htop / top: sanity checks for CPU and thread behavior
- wrk (or similar): load generation for HTTP server benchmarks

Evaluation Methodology

- Use the same workloads and parameters across all implementations
- Run multiple trials and report the averaged results
- Control experimental conditions (same machine, compiler flags, thread counts)

Expected Graphs

- Line charts of throughput vs. concurrency to show scalability across thread pool designs and coroutines
- Line charts of tail latency (p95/p99) vs. concurrency to capture overload and scheduling effects
- Line charts of CPU utilization vs. concurrency to compare efficiency and idle time
- Line charts of context switches vs. concurrency to highlight scheduling and synchronization overhead

References

[1] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 53–79.

[2] BELSON, B., HOLDSWORTH, J., XIANG, W., AND PHILIPPA, B. A survey of asynchronous programming using coroutines in the internet of things and embedded systems. *ACM Transactions on Embedded Computing Systems (TECS) 18*, 3 (2019), 1–21.

[3] BELSON, B., XIANG, W., HOLDSWORTH, J., AND PHILIPPA, B. C++ 20 coroutines on microcontrollers—what we learned. *IEEE Embedded Systems Letters 13*, 1 (2020), 9–12.

[4] BERONIĆ, D., MODRIĆ, L., MIHALJEVIĆ, B., AND RADOVAN, A. Comparison of structured concurrency constructs in java and kotlin–virtual threads and coroutines. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)* (2022), IEEE, pp. 1466–1471.

[5] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM) 46*, 5 (1999), 720–748.

[6] CHASE, D., AND LEV, Y. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures* (2005), pp. 21–28.

[7] HOLT, R. C. Some deadlock properties of computer systems. *ACM Computing Surveys (CSUR) 4*, 3 (1972), 179–196.

[8] HUANG, K., WANG, T., ZHOU, Q., AND MENG, Q. The art of latency hiding in modern database engines. *Proceedings of the VLDB Endowment 17*, 3 (2023), 577–590.

[9] JONATHAN, C., MINHAS, U. F., HUNTER, J., LEVANDOSKI, J., AND NISHANOV, G. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment 11*, 11 (2018), 1702–1714.

[10] LEA, D. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (2000), pp. 36–43.

[11] LEE, E. A. The problem with threads. *Computer 39*, 5 (2006), 33–42.

[12] LIAO, G., GUO, D., BHUYAN, L., AND KING, S. R. Software techniques to improve virtualized i/o performance on multi-core systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2008), pp. 161–170.

[13] LING, Y., MULLEN, T., AND LIN, X. Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review 34*, 2 (2000), 42–55.

[14] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: an efficient and portable web server. In *USENIX Annual Technical Conference, General Track* (1999), pp. 199–212.

[15] PERRON, M., CASTRO FERNANDEZ, R., DEWITT, D., CAFARELLA, M., AND MADDEN, S. Cackle: Analytical workload cost and performance stability with elastic pools. *Proceedings of the ACM on Management of Data 1*, 4 (2023), 1–25.

[16] STROUSTRUP, B. *A Tour of C++*. Addison-Wesley Professional, 2022.

[17] TORNG, C., WANG, M., AND BATTEN, C. Asymmetry-aware work-stealing runtimes. *ACM SIGARCH Computer Architecture News 44*, 3 (2016), 40–52.

[18] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable threads for internet services. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 268–281.

[19] WELSH, M., CULLER, D., AND BREWER, E. Seda: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS operating systems review 35*, 5 (2001), 230–243.

[20] XU, X., AND LI, G. Research on coroutine-based process interaction simulation mechanism in c++. In *Asian Simulation Conference* (2012), Springer, pp. 178–187.