Kernel-Level and User-Level Thread Implementation: A Quantitative Comparative Analysis on ARM64 Linux

Jin Zemin
Department of Data Science
The Chinese University of Hong Kong, Shenzhen
225040149@link.cuhk.edu.cn

Zou Haonan
Department of Data Science
The Chinese University of Hong Kong, Shenzhen
225040132@link.cuhk.edu.cn

Abstract*—Operating systems expose two fundamental threading interfaces: user-level threads (ULT) implemented via POSIX pthreads and kernel-level threads (KLT) implemented via the Linux `kthread` API. Although both run atop the same Completely Fair Scheduler (CFS), they differ in privilege level, scheduling visibility, and the cost of concurrent access to shared state. This paper presents a controlled, five-scenario experiment on an ARM64 Ubuntu 24.04 system running Linux 6.8 inside a UTM virtual machine with four vCPUs. Each scenario systematically varies two independent variables: (1) the number of CPUs to which four worker threads are pinned (4, 2, or 1), and (2) the scheduling-intervention mechanism applied every 10 000 iterations (none, `sched_yield`/`schedule()`, or `usleep(1)`/`usleep_range(1,2)`). We measure throughput in mega-operations per second (Mops/s) and per-thread voluntary and involuntary context-switch counts over five independent runs per scenario. Our results show three principal findings. First, `sched_yield()` keeps threads in the `TASK_RUNNING` state and therefore never generates a voluntary context switch (vol_csw $= 0$ exactly), whereas `usleep(1)` transitions threads to `TASK_INTERRUPTIBLE` and generates approximately 100 voluntary switches per thread per run. Second, collapsing four threads from four CPUs to one CPU raises throughput by $2.9\times$ for ULT and $3.2\times$ for KLT, because a single CPU keeps the shared cache line in the `Modified` state permanently, eliminating the MESI cross-core invalidation traffic that costs 100–300 cycles per atomic operation under multi-CPU contention. Third, at baseline (zero scheduling intervention, four CPUs) KLT achieves 109.5 Mops/s versus ULT's 87.3 Mops/s, a 25.4% advantage attributable to the absence of user-to-kernel mode transitions during the hot loop and to direct `task_struct` field access for context-switch accounting. The implementation code and raw data are presented in full, grounding every performance claim in observable kernel mechanisms.

***Index Terms*—kernel threads, user-level threads, POSIX pthreads, Linux kthread, context switches, MESI protocol, cache coherence, CPU affinity, CFS scheduler, ARM64**

I. Introduction and Motivation

Threads are the fundamental unit of CPU utilization in modern operating systems. Linux exposes two principally distinct threading models to application and system programmers. User-Level Threads (ULT), realized through the Native POSIX Thread Library (NPTL) via `pthread_create()`, execute in user space (ring 3) and request kernel services through system calls. Kernel-Level Threads (KLT), created with `kthread_create()`, execute directly in kernel space (ring 0) and are managed and scheduled by the kernel without any privilege-level boundary crossing during their hot path.

Despite sharing the same underlying CFS scheduling policy and the same 1:1 mapping of threads to kernel scheduling entities, the two models differ in measurable ways. Three concrete questions motivate this study:

- **Q1 (Scheduling classification):** How does Linux distinguish voluntary from involuntary context switches at the kernel level, and does the distinction between `sched_yield()` and `usleep()` lead to different scheduling behaviors?
- **Q2 (CPU contention and cache coherence):** When multiple CPUs simultaneously operate on a single shared atomic counter, does adding more CPUs always improve aggregate throughput?
- **Q3 (KLT vs. ULT performance gap):** Where exactly does the performance difference between KLT and ULT originate at the code and architecture level?

To answer these questions, we design a structured five-scenario experiment using two parallel codebases: `uthread_demo.c` (ULT via pthreads) and `kthread_demo.c` (KLT as a loadable kernel module). Both codebases implement the same logical experiment—four worker threads pinned to varying numbers of CPUs, executing one million atomic increments each, with optional scheduling interventions every 10,000 iterations—enabling a controlled, apples-to-apples comparison.

II. Background and Related Work

Actually, the distinction between user-level threads (ULTs) and kernel-level threads (KLTs) has long been studied in operating systems research. Early work by Thomas E. Anderson et al. introduced scheduler activations, showing that purely user-level thread management suffers from limited kernel visibility, especially under blocking and multiprocessor scheduling scenarios [1]. This motivates kernel-assisted scheduling models used in modern systems.

In Linux systems, thread scheduling is fully handled by the kernel's Completely Fair Scheduler (CFS), and thread context-

switching costs are tightly coupled with kernel scheduling behavior. These mechanisms and their performance implications are described in detail in Robert Love's Linux Kernel Development [2], which provides the basis for understanding kernel-level threading overhead and scheduling behavior.

We now provide a more detailed discussion of Linux thread models and context-switch behavior:

A. Thread Models in Linux

Linux implements a unified scheduling architecture in which both processes and threads are represented by a `task_struct` and scheduled identically by CFS [2]. A pthread created with `pthread_create()` results in a `clone()` system call with shared memory flags; all resulting threads share a single process ID (PID) but each receives a unique thread ID (TID) from `gettid()`. A kthread created with `kthread_create()` results in an independent `task_struct` whose PID equals its TID—it appears as a separate process in `ps(1)` output.

B. Context Switch Classification

The Linux kernel records two disjoint context-switch counters per task in `task_struct`: `nvcsw` (voluntary context switches) and `nivcsw` (involuntary context switches) [3]. A voluntary context switch occurs when a task transitions to a non-runnable state (e.g., `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`) and explicitly calls into the scheduler, for instance through `nanosleep()` or `wait_event()`. An involuntary context switch occurs when the kernel preempts a task that remains `TASK_RUNNING`—either because its time slice expired or because a higher-priority task became runnable.

The critical distinction for this experiment is that `sched_yield()` and its kernel-space equivalent `schedule()` called while the thread is still `TASK_RUNNING` are recorded as *involuntary* switches, not voluntary ones. The thread never gives up ownership of a runnable state; it merely moves to the back of the run queue.

C. MESI Cache Coherence Protocol

ARM64 processors implement cache coherence mechanisms at the L1 and L2 levels, typically based on MESI/MOESI-like protocols. [4]. When a cache line is in the `Modified` state on one core and another core attempts a read or read-modify-write of the same address, the holding core must write back the line and transition it to the `Shared` or `Invalid` state. For atomic read-modify-write instructions (`LDADD` on ARM64, `LOCK XADD` on x86), every successful completion invalidates the cache line on all other cores. This invalidation requires communication across the inter-core coherence interconnect, incurring tens to hundreds of clock cycles on modern out-of-order processors, compared to a few cycles for an L1 cache hit.

D. Atomic Operations

The GCC built-in `__sync_fetch_and_add()` used in `uthread_demo.c` compiles to a single `LDADD` instruction on ARM64, which is a hardware-atomic load-add with full memory barrier semantics. The kernel macro `atomic_inc()` used in `kthread_demo.c` expands to the same `LDADD` instruction via `linux/atomic.h` architecture abstractions [5]. The correctness of the final counter value (`NUM_THREADS`×`ITERATIONS`= 4,000,000) is therefore guaranteed in all scenarios without additional locking.

III. Design and Implementation

A. Experimental Framework Overview

Both `uthread_demo.c` and `kthread_demo.c` implement the same logical five-scenario matrix. Each scenario is a two-dimensional point in the space (CPUs used × intervention type), as summarized in Table I.

TABLE I
Five Experimental Scenarios

ID	Label	CPUs	Intervention
A	Baseline	4	None
B1	Mod. contention	2	`sched_yield()` / `schedule()`
B2	Max. contention	1	`sched_yield()` / `schedule()`
C1	Voluntary, mod.	2	`usleep(1)` / `usleep_range(1,2)`
C2	Voluntary, max.	1	`usleep(1)` / `usleep_range(1,2)`

All four threads execute 10^6 iterations of an atomic shared-counter increment. Every 10,000 iterations (i.e., at loop index i where $i \bmod 10{,}000 = 0$), the configured intervention is applied, producing approximately 100 intervention events per thread per run. After all threads complete, throughput is computed as:

$$\text{Throughput} = \frac{N_{\text{threads}} \times \text{ITERATIONS}}{\Delta t_{\text{wall}}} \text{ [ops/s]}$$

and expressed in Mops/s for readability.

B. CPU Affinity Binding

The CPU assignment formula is identical in both implementations:

$$\text{target\_cpu}(i) = i \bmod \text{cpus\_used}$$

giving: threads $(0, 1, 2, 3)$ map to CPUs $(0, 1, 2, 3)$ for four CPUs; to $(0, 1, 0, 1)$ for two CPUs; and to $(0, 0, 0, 0)$ for one CPU.

In ULT, affinity is set inside each worker thread using `pthread_setaffinity_np()` after `pthread_create()` returns:

```
1  cpu_set_t cpuset;
2  CPU_ZERO(&cpuset);
3  CPU_SET(t->cpu_id, &cpuset);
4  pthread_setaffinity_np(pthread_self(),
5      sizeof(cpu_set_t), &cpuset);
```

Listing 1. ULT CPU affinity (uthread_demo.c, lines 149–154)

In KLT, affinity *must* be bound before `wake_up_process()` is called; binding after wakeup races with the scheduler:

```
int target_cpu = i % cpus_used;
kthread_bind(threads[i], target_cpu);
wake_up_process(threads[i]); /* AFTER bind */
```

Listing 2. KLT CPU affinity (kthread_demo.c, lines 319–321)

The ordering `kthread_bind()` → `wake_up_process()` is critical: `kthread_bind()` sets the `PF_NO_SETAFFINITY` flag and records the target CPU in the task, then `wake_up_process()` enqueues the thread on exactly that CPU's run queue. Reversing the order could allow the scheduler to place the kthread on an arbitrary CPU before the binding takes effect.

C. Barrier Synchronization

Barrier synchronization ensures that all threads begin executing the hot loop simultaneously. Without a barrier, fast threads could complete a significant fraction of their iterations before slower threads even start, eliminating the intended concurrent contention in scenarios B1, B2, C1, and C2.

ULT barrier uses `pthread_barrier_t`:

```
pthread_barrier_init(&start_barrier, NULL,
                     NUM_THREADS);
/* in each thread: */
pthread_barrier_wait(&start_barrier);
```

Listing 3. ULT barrier (uthread_demo.c, lines 251, 160)

KLT barrier implements a spin barrier using an atomic counter and `schedule()`, because pthreads primitives are not available in kernel space:

```
atomic_inc(&barrier_count);
while (atomic_read(&barrier_count) < num_threads)
    schedule();
```

Listing 4. KLT spin barrier (kthread_demo.c, lines 181–183)

Each kthread atomically increments `barrier_count` and then spins—yielding the CPU each iteration of the spin—until all `num_threads` threads have checked in. The `schedule()` call inside the barrier spin serves a critical purpose: it prevents any single kthread from monopolizing a CPU during the wait phase, while still leaving the thread in `TASK_RUNNING` state (not sleeping). This spin does introduce a small number of involuntary context switches before the timed hot loop begins; those switches are excluded from the measurement because the CSW snapshot is taken *after* the barrier releases and immediately before `ktime_get()` marks the start of the timing window.

D. Atomic Shared Counter

ULT: The shared counter is a `volatile int` incremented with the GCC compiler built-in:

```
__sync_fetch_and_add(&shared_counter, 1);
```

Listing 5. ULT atomic increment (uthread_demo.c, line 175)

The `volatile` qualifier prevents the compiler from caching the variable in a register across loop iterations; `__sync_fetch_and_add()` emits a hardware atomic instruction (`LDADD` on ARM64) with full acquire-release memory barrier semantics, guaranteeing that the final value of `shared_counter` equals 4,000,000 regardless of thread interleaving.

KLT: The shared counter uses the kernel's `atomic_t` type:

```
static atomic_t shared_counter = ATOMIC_INIT(0);
/* in hot loop: */
atomic_inc(&shared_counter);
```

Listing 6. KLT atomic increment (kthread_demo.c, lines 123, 196)

`atomic_t` wraps an `int` in a structure to prevent accidental non-atomic access; `atomic_inc()` expands to the same `LDADD` instruction on ARM64 via the kernel's architecture-specific atomic operations header.

E. Intervention Mechanisms: sched_yield/schedule() vs. usleep

The two intervention mechanisms are the core of the experiment's independent variables. Their behavior differs fundamentally at the scheduler level.

1) sched_yield() (ULT) and schedule() (KLT) — B group: `sched_yield()` in user space issues a `sched_yield` system call, which invokes `do_sched_yield()` in the kernel. `schedule()` called directly in kthread code takes the same path without the system call overhead. In both cases the thread's state remains `TASK_RUNNING`. The kernel moves the thread to the back of its CPU's CFS run queue (updating its virtual runtime to the current minimum in the queue) and immediately selects the next eligible task. Because the task never left the runnable state, the kernel increments `current->nivcsw` (non-voluntary context switch counter) and does *not* touch `current->nvcsw`. Therefore **vol_csw is guaranteed to be exactly 0** across all B-group runs:

```
case YIELD:
    /* thread stays TASK_RUNNING */
    schedule();   /* increments nivcsw */
    break;
```

Listing 7. B-group intervention (kthread_demo.c, lines 201–215)

In Scenario B1 (2 threads per CPU), each `schedule()` finds one peer on the same CPU's run queue and switches to it, then switches back. In Scenario B2 (4 threads on one CPU), the scheduler rotates through all three peers before returning to the calling thread, which means longer waits per intervention but similar total switch counts because each thread is also yielding.

2) usleep(1) (ULT) and usleep_range(1,2) (KLT) — C group: `usleep(1)` in user space calls `nanosleep()` with $t_{\text{min}} = 1\,\mu$s. The kernel transitions the thread to `TASK_INTERRUPTIBLE`, removes it from the run queue, arms a high-resolution timer (`hrtimer`) for $1\,\mu$s, and runs the scheduler. When the timer fires, the callback re-enqueues the thread. Because the thread explicitly blocked, the kernel increments `current->nvcsw` upon the context switch out. Therefore **vol_csw** rises by approximately one per call, and

with 100 calls per run per thread the expected total is $\approx 100 \times 4 = 400$ across all threads per scenario, matching the observed 399.4 ± 0.9 (ULT C1) and 399.2 ± 0.8 (ULT C2).

In kernel space, `usleep()` is a libc function and does not exist. `usleep_range(1, 2)` from `<linux/delay.h>` is the direct kernel equivalent; it calls `hrtimer_nanosleep()` internally and transitions the thread to `TASK_INTERRUPTIBLE` in exactly the same manner:

```
case SLEEP:
    /* TASK_INTERRUPTIBLE -> increments nvcsw */
    usleep_range(1, 2);
    break;
```

Listing 8. C-group intervention (kthread_demo.c, lines 218–235)

F. Context-Switch Measurement

ULT: Context switches are read from the Linux `procfs` interface. Because all pthreads in a process share the same PID, reading `/proc/self/status` would give the process-wide aggregate. The implementation therefore uses the per-thread path `/proc/self/task/<TID>/status`, obtaining the TID via `syscall(SYS_gettid)`:

```
static void read_csw(long *vol, long *invol) {
    pid_t tid = (pid_t)syscall(SYS_gettid);
    char path[64];
    snprintf(path, sizeof(path),
        "/proc/self/task/%d/status", tid);
    /* parse voluntary_ctxt_switches and
       nonvoluntary_ctxt_switches fields */
}
```

Listing 9. ULT per-thread CSW read (uthread_demo.c, lines 121–138)

KLT: Context switches are read directly from the task's `task_struct` fields without any system call, since kthread code already executes in kernel space:

```
long vol_before   = current->nvcsw;
long invol_before = current->nivcsw;
/* ... hot loop ... */
current->nvcsw  - vol_before  /* delta vol */
current->nivcsw - invol_before /* delta inv */
```

Listing 10. KLT CSW read (kthread_demo.c, lines 187–188, 255–256)

This direct `task_struct` access is zero-overhead compared to the ULT approach of opening and parsing a `/proc` file for every measurement point. Although the measurements are only taken once before and once after the hot loop (not inside it), the ULT's `/proc` parsing introduces two `fopen()`, `fgets()`, and `fclose()` system calls per thread per scenario run, whereas KLT reads are simple struct field dereferences.

G. Thread Identity and Scheduling

A key architectural difference between ULT and KLT is their identity as seen by the kernel scheduler.

ULT pthreads: All four pthreads share the process PID (e.g., 1401) but hold unique TIDs (1402–1405). The `task_struct` instances are grouped under the same `mm_struct` (shared virtual address space). The Linux CFS scheduler sees each TID as an independent scheduling entity in a 1:1 mapping.

KLT kthreads: Each kthread has its own unique PID that equals its TID (e.g., 1767, 1768, 1769, 1770). There is no separate TID namespace; the kthread PID *is* the kernel task identifier. Each kthread has an independent `task_struct` with no shared `mm_struct` (kernel threads use the `init_mm` global memory descriptor for any kernel memory access they perform, but they do not map user-space pages). The CFS scheduler treats these four task_structs as four independent scheduling entities, with no difference in scheduling policy compared to ULT pthreads.

H. ARM64-Specific Kernel Considerations

The ARM64 kernel is compiled with the `-mgeneral-regs-only` flag, which prohibits the use of floating-point or SIMD registers in kernel code. Any format string containing `%f` or `%lf` in a `printk()` call causes a compile-time error. `kthread_demo.c` resolves this by performing all throughput and timing formatting using pure integer arithmetic:

```
/* "X.YY ms": */
*whole = ns / 1000000LL;
*frac  = (ns % 1000000LL) / 10000LL;
/* "X.Y Mops/s": */
*whole = ops / 1000000LL;
*frac  = (ops % 1000000LL) / 100000LL;
```

Listing 11. ARM64-safe integer formatting (kthread_demo.c, lines 144–160)

IV. Experimental Setup and Methodology

A. Hardware and Software Environment

TABLE II
Experimental Environment

Parameter	Value
Host hardware	Apple Silicon (M-series)
Virtualization	UTM (QEMU-based)
Guest OS	Ubuntu 24.04 (aarch64)
Kernel version	6.8.0-101-generic
Architecture	ARM64
Virtual CPUs	4 (`nproc`= 4)
Compiler (ULT)	gcc-13, flags: `-O0 -lpthread`
Compiler (KLT)	aarch64-linux-gnu-gcc-13
KLT build	Kernel module (`.ko`)
ULT interface	pthreads (NPTL, glibc)
KLT interface	Linux `kthread` API

The `-O0` flag is deliberately used for the ULT build to prevent the compiler from optimizing away the shared-counter write or hoisting the `volatile` read out of the loop, ensuring the benchmark measures real atomic bus traffic rather than a dead-code-eliminated loop.

B. Run Procedure

ULT: A single invocation of `./uthread_demo` runs all five scenarios sequentially with a one-second sleep between scenarios to allow OS resource cleanup. Each scenario resets `shared_counter=0` and reinitializes the barrier. This single-binary sequential design ensures identical process environment across scenarios.

KLT: Each scenario requires loading the module with the `scenario=N` parameter, waiting for threads to complete their hot loop (threads then block in `kthread_should_stop()` polling with `schedule_timeout_interruptible(100ms)`), then unloading the module:

```
sudo insmod kthread_demo.ko scenario=N
sudo dmesg | grep KLT_LAB
sudo rmmod kthread_demo
```

Listing 12. KLT run procedure

Repetitions: Each scenario is run $n = 5$ independent times. Results are reported as Mean $\pm$ SD using Bessel's correction. The coefficient of variation (CV) is used to assess measurement stability.

C. Timing Methodology

ULT: Each thread records its individual start and end times using `clock_gettime(CLOCK_MONOTONIC)`. The wall time for the scenario is the outer join time from `pthread_join()` calls. Throughput uses the wall time because all threads start synchronously at the barrier and finish at different times depending on the intervention; the latest-thread finish time upper-bounds the real elapsed time.

KLT: Each kthread records `ktime_get()` before and after its hot loop into per-thread arrays. The module exit function computes elapsed time as $\max(\text{end}_i) - \min(\text{start}_i)$, explicitly excluding thread-creation overhead and the initial barrier spin from the measurement window.

V. Results and Analysis

A. Raw Aggregated Results

Table III and Table IV present the aggregated ULT and KLT results respectively, each computed over five independent runs. Context-switch counts are per-thread aggregated totals (sum over 4 threads).

TABLE III
ULT (PTHREAD) AGGREGATED RESULTS ($n = 5$ RUNS). MEAN $\pm$ SD.

Scenario	Int.	vol_csw	inv_csw	Thrpt (Mops/s)	CV
A (4T/4CPU)	None	0	12.6 ± 2.7	87.3 ± 3.0	3.4%
B1 (4T/2CPU)	yield	0	302.4 ± 12.6	93.9 ± 4.0	4.3%
B2 (4T/1CPU)	yield	0	392.8 ± 10.6	250.3 ± 9.2	3.7%
C1 (4T/2CPU)	usleep(1)	399.4 ± 0.9	103.0 ± 6.5	144.1 ± 5.7	4.0%
C2 (4T/1CPU)	usleep(1)	399.2 ± 0.8	10.8 ± 2.4	165.4 ± 5.7	3.5%

TABLE IV
KLT (KTHREAD) AGGREGATED RESULTS ($n = 5$ RUNS). MEAN $\pm$ SD.

Scenario	Int.	vol_csw	inv_csw	Thrpt (Mops/s)	CV
A (4T/4CPU)	None	0	0	109.5 ± 3.0	2.7%
B1 (4T/2CPU)	schedule	0	356.0 ± 13.3	171.1 ± 6.1	3.6%
B2 (4T/1CPU)	schedule	0	282.6 ± 11.4	349.9 ± 9.0	2.6%
C1 (4T/2CPU)	usleep_range	398.8 ± 1.3	8.0 ± 1.6	185.2 ± 17.5	9.4%
C2 (4T/1CPU)	usleep_range	399.2 ± 0.8	0	163.0 ± 3.9	2.4%

B. Q1 Analysis: Voluntary vs. Involuntary Context Switches

1) B group (sched_yield / schedule): vol_csw = 0 always: In every B-group run for both ULT and KLT, `vol_csw` is exactly zero. This is not a measurement artifact; it follows directly from the kernel's classification logic. When `sched_yield()` is called in user space, it issues the `sys_sched_yield` system call, which executes `do_sched_yield()` in the kernel. This function calls `__schedule(SM_NONE)` with the current task's state already `TASK_RUNNING` and without setting any wait flag. Since the task never transitions to a blocked state, the kernel increments `nivcsw` (the involuntary counter) and leaves `nvcsw` unchanged. The ULT implementation observes this via delta reads of `/proc/self/task/<TID>/status`; the KLT implementation observes it by reading `current->nvcsw` directly.

In ULT Scenario B1 (4T/2CPU), `inv_csw` reaches 302.4 ± 12.6; in B2 (4T/1CPU), it reaches 392.8 ± 10.6. The higher inv_csw in B2 than B1 is explained by the greater queue depth: with four threads contending for one CPU, each `schedule()` call may rotate through multiple peers before the original thread is rescheduled, increasing the total switch count. The absolute values exceed the theoretical minimum of 100 (one switch per 10,000-iteration block) because additional preemption events occur during the hot loop outside the explicit intervention points.

2) C group (usleep / usleep_range): vol_csw $\approx$ 100 per thread: In all C-group runs, `vol_csw` per thread converges tightly to approximately 100, giving a four-thread total of $\approx$ 400. This is explained by the sleep mechanism: `usleep(1)` calls `nanosleep()`, which sets the thread's state to `TASK_INTERRUPTIBLE` before calling `schedule()`. Because the thread is not `TASK_RUNNING` when the context switch occurs, the kernel increments `nvcsw`. The number of calls is $\lfloor 10^6/10{,}000 \rfloor = 100$, and since the timer fires for each call, vol_csw tracks the intervention count almost exactly. The sub-unit deviations (e.g., 399.4 ± 0.9 vs. the theoretical 400.0) arise from edge cases at loop boundaries where $i \bmod 10{,}000 = 0$ is reached at $i = 0$ (the first iteration), and the corresponding sleep may partially overlap with the barrier-exit phase in some runs.

The coexistence of `inv_csw` $= 103.0 \pm 6.5$ in ULT C1 is also noteworthy: even with usleep-based voluntary yielding, the OS occasionally preempts threads involuntarily—for instance, to run OS background tasks—adding a non-zero inv_csw count. In KLT C1, inv_csw is only 8.0 ± 1.6, suggesting the kernel scheduler is less aggressive about preempting kernel

threads than user-space threads, consistent with kernel threads' historically higher scheduling priority under CFS.

C. Q2 Analysis: MESI Cache Coherence and Single-CPU Throughput Surge

The most counterintuitive result in the experiment is that consolidating four threads from four CPUs onto one CPU dramatically *increases* throughput. Concretely:

- ULT: A (4CPU) achieves 87.3 Mops/s; B2 (1CPU) achieves 250.3 Mops/s—a $2.87\times$ improvement.
- KLT: A (4CPU) achieves 109.5 Mops/s; B2 (1CPU) achieves 349.9 Mops/s—a $3.20\times$ improvement.

The explanation lies in the MESI cache coherence protocol and the cost of atomic operations across cores.

Scenario A (4 CPUs): Each of the four CPUs maintains its own L1 cache. When CPU 0 executes `atomic_inc(&shared_counter)` (LDADD), it requests exclusive ownership of the cache line holding `shared_counter`. If CPU 1 holds that line in `Modified` or `Exclusive` state, it must first write back the line and transition it to `Invalid`. CPU 0 then reads the updated value from L3 or directly from CPU 1's L1 (depending on the coherence topology) and performs the atomic add. On the ARM64 Cortex-A architecture implemented in Apple Silicon, this cross-core round trip costs approximately 100–300 clock cycles. With four cores competing in round-robin for the same 8-byte cache line, the hot loop is almost entirely stalled on cache-line ownership negotiation rather than actual computation.

Scenario B2/C2 (1 CPU): All four threads are pinned to CPU 0. The OS time-slices them cooperatively (via `schedule()`) or by timer preemption. At any point in time, only one thread runs. Between context switches, the `shared_counter` cache line remains in the `Modified` state in CPU 0's L1 cache. There is never a competing CPU requesting ownership, so every `LDADD` hits L1 at a cost of 4–10 cycles. The $\sim 3\times$ speedup is therefore the ratio of cross-core atomic cost ($\sim$ 200 cycles) to L1 atomic cost ($\sim$ 7 cycles), which matches the observed throughput ratio.

This finding directly answers Q2: more CPUs *degrade* throughput when the workload consists of repeated atomic operations on a single shared variable. Parallelism only improves performance when work is partitioned such that each thread's working set is private to its cache.

D. Q3 Analysis: KLT vs. ULT Performance Gap

Table V presents the side-by-side throughput comparison.

1) Baseline KLT advantage (+25.4% in Scenario A): Two code-level mechanisms account for KLT's baseline advantage:

(1) Zero user-kernel mode transitions in the hot loop. In the ULT hot loop, every call to `__sync_fetch_and_add()` is a pure user-space instruction—no system call is needed for atomic operations. However, the context-switch measurement (`read_csw()`) does require a `syscall(SYS_gettid)`, an `open()`, multiple `fgets()`, and a `fclose()`, all of which are system calls. More importantly, the hot loop itself operates in ring 3, meaning every hardware interrupt or timer tick that the kernel needs to service requires a privilege-level save of the current user-space register state. KLT threads operate in ring 0; hardware interrupts are handled without any privilege-level transition, and there is no user-kernel boundary cost during the main computation.

TABLE V
KLT VS. ULT THROUGHPUT COMPARISON (MOPS/S, MEAN $\pm$ SD)

Scenario	ULT	KLT	KLT gain
A (4T/4CPU, none)	87.3 ± 3.0	109.5 ± 3.0	$+25.4\%$
B1 (4T/2CPU, yield)	93.9 ± 4.0	171.1 ± 6.1	$+82.2\%$
B2 (4T/1CPU, yield)	250.3 ± 9.2	349.9 ± 9.0	$+39.8\%$
C1 (4T/2CPU, usleep)	144.1 ± 5.7	185.2 ± 17.5	$+28.5\%$
C2 (4T/1CPU, usleep)	165.4 ± 5.7	163.0 ± 3.9	-1.5%

(2) Direct `task_struct` field access for CSW accounting. When the hot loop ends, the KLT worker reads `current->nvcsw` directly, a single pointer dereference. The ULT worker must call `open()` on `/proc/self/task/<TID>/status`, read the file into a line buffer, and parse two integers with `sscanf()`. Although this measurement cost is excluded from the hot loop timing, it contributes to overall execution overhead that the operating system must service on behalf of ULT threads.

2) Large KLT advantage in B1 (+82.2%): The outsized advantage of KLT in Scenario B1 (4T/2CPU, schedule()) reflects a scheduling efficiency difference. In the ULT case, each `sched_yield()` involves a system call round trip: user space $\rightarrow$ kernel (ring 3 $\rightarrow$ ring 0) $\rightarrow$ scheduler decision $\rightarrow$ return to user space. In the KLT case, each `schedule()` is a direct kernel function call with no privilege transition. With 100 yield events per thread and 2 CPUs each hosting 2 competing threads, the ULT accumulates 400 mode-switch penalties per scenario run, whereas the KLT incurs zero. At approximately 1,000–3,000 cycles per mode switch on ARM64, this deficit compounds significantly.

3) Convergence in C2 (-1.5%): In Scenario C2 (4T/1CPU, usleep/usleep_range), the KLT advantage essentially vanishes (163.0 vs. 165.4 Mops/s, within measurement noise). This convergence occurs because the dominant cost in C2 is the 1 μs sleep latency plus the hrtimer wakeup path, which costs approximately the same whether invoked from user space (`usleep(1)` $\rightarrow$ `nanosleep()` system call) or from kernel space (`usleep_range(1,2)` $\rightarrow$ `hrtimer_nanosleep()`). Both paths ultimately block in the same kernel sleep queue, armed by the same hrtimer subsystem. The per-call overhead of the system call is amortized over the 1 μs minimum sleep, making the syscall cost negligible. Additionally, the higher CV in KLT C1 (9.4%) relative to all other scenarios reflects variability in hrtimer wakeup latency when two threads compete on 2 CPUs—a known source of jitter in high-resolution timer workloads on virtualized ARM64.

4) Thread identity summary:

TABLE VI
THREAD IDENTITY: ULT VS. KLT

	PID	TID	task_struct	Sched.
ULT	Shared (e.g. 1401)	Unique (1402–1405)	4 per process	CFS (1:1)
KLT	Unique (e.g. 1767–1770)	PID = TID	4 independent	CFS (1:1)

VI. DISCUSSION

A. The Paradox of Parallelism on Shared Atomic State

The experiment provides a concrete quantitative demonstration of Amdahl's Law's parallelism ceiling. When the entire workload is a sequential bottleneck—a single atomic shared counter—additional CPUs provide zero benefit and actively harm performance due to cache coherence overhead. The practical implication for system design is that atomic operations on shared scalars should be partitioned (e.g., per-CPU counters merged at read time) when high throughput is required. Linux's `percpu_counter` and `local_t` abstractions exist precisely for this reason.

B. sched_yield() as an Anti-Pattern for Cooperative Yielding

A common misconception among developers is that `sched_yield()` is a cooperative yielding primitive that allows other threads to run. This experiment quantitatively confirms that it is not. In Scenario B1 (2 threads per CPU), `sched_yield()` does cause the thread to be rescheduled—but only because there is actually a competing peer on the same CPU's run queue. In scenario A, where each thread has a dedicated CPU, `sched_yield()` effectively becomes a no-op: the scheduler finds no other runnable task on that CPU and immediately returns the running thread. Furthermore, `sched_yield()` is classified as *involuntary* by the kernel, which has implications for scheduler accounting and priority boosting heuristics in CFS.

For I/O-bound cooperative yielding where the intent is to truly give up the CPU and allow timer-driven wakeup, `usleep()` (user space) or `usleep_range()` (kernel space) are the correct primitives.

C. KLT Performance Is Not Uniformly Superior

While KLT outperforms ULT in most scenarios, the Scenario C2 convergence demonstrates that the advantage disappears when the dominant cost is a kernel subsystem operation (hrtimer sleep) that both threading models invoke with similar overhead. KLT's advantage is specifically located at the user-kernel boundary: mode transitions, procfs file I/O, and signal handling overhead. For workloads that are purely compute-bound or that spend significant time in kernel waits, the KLT vs. ULT distinction becomes operationally irrelevant.

D. Measurement Methodology Validity

The barrier synchronization design is critical to result validity. Without simultaneous start, early-finishing threads could release their CPU pinning pressure, reducing contention for late-starting threads and systematically underestimating inv_csw in B-group scenarios. The KLT atomic barrier introduces a bounded spin overhead before the measurement window, but because CSW snapshots are taken *after* the barrier and timing starts simultaneously, this overhead is excluded from all reported metrics.

The five-repetition design with CV reporting allows assessment of measurement stability. All CVs are below 5% except KLT C1 (9.4%), which we attribute to hrtimer latency jitter in the virtualized environment. The counter-verification check (confirming `shared_counter`$= 4{,}000{,}000$ in all 25 per-scenario thread measurements for both ULT and KLT) provides strong evidence that the atomic operations are correctly implemented and no increments are lost to data races.

E. Limitations

Several factors limit the generalizability of these results. First, the experiment runs inside a UTM virtual machine, meaning the "CPUs" are vCPUs, and the host hypervisor may introduce additional scheduling jitter that inflates inv_csw counts and adds variance to timing measurements. Second, the ARM64 Apple Silicon MESI implementation may have different L1/L2/L3 latency characteristics than server-class Arm Neoverse or x86 Intel/AMD CPUs, and the observed $3\times$ single-CPU speedup may not generalize. Third, the use of `-O0` for the ULT build ensures measurement fidelity but does not represent production code; a production-optimized build might yield different relative performance due to compiler vectorization or loop transformations.

VII. ADVANCED OPTION: USER-SPACE COROUTINES

A. Experiment Introduction

The purpose of this experiment is to master the basic implementation principles of user state coroutines, understand the mechanism and process of context switching, analyze the effectiveness of coroutine scheduling strategies, and compare the differences between user state coroutines and kernel threads.

Coroutine is a lightweight thread in user mode, characterized by being controlled and scheduled by the user program rather than the operating system kernel, resulting in low context switching overhead and no need to get stuck in kernel mode. Shared memory space among coroutines within the same process is particularly suitable for handling I/O-intensive tasks.

This experiment uses the ‘ ucontext. h ‘ library provided by POSIX to implement context switching, mainly through three key functions: ‘ getcontext() ‘ to obtain the current context, ‘ makecontext() ‘ to set the execution function and stack space of the context, and ‘ swapcontext() ‘ to save the current context and switch to the new context. The coroutine requires independent stack space to store local variables and function call information. In this experiment, a global array is pre allocated as stack

space. In practical projects, it is recommended to use 'malloc()' dynamic allocation to improve flexibility.

B. Implementation

The execution process of experimental code can be divided into the following stages: firstly, the program initializes the main context 'main_ctx' and the coroutine context 'thread_ctx', and allocates stack space for the coroutine. Then, associate the coroutine context with the execution function 'thread_func()' through 'makecontext()'. After the main program starts, the first call to 'swapcontext()' switches to coroutine execution. During the execution of the coroutine, the business logic is simulated through a loop, and after each loop, 'swapcontext()' is actively called to switch back to the main program. After each return from the coroutine, the main program performs some work and then switches back to the coroutine again. After the execution of the coroutine is completed, it ultimately switches back to the main program, which prints the end message and exits.

```
void thread_func() {
    printf("[Thread] I am running in the user-
        space thread!\n");

    for (int i = 0; i < 3; i++) {
        printf("[Thread] Iteration %d\n", i);
        sleep(1);
        swapcontext(&thread_ctx, &main_ctx);
    }

    printf("[Thread] Thread finished. Switching
        back to main permanently.\n");

    swapcontext(&thread_ctx, &main_ctx);
}
```

Listing 13. coroutine_demo.c(coroutine task)

```
thread_ctx.uc_stack.ss_sp = thread_stack;
thread_ctx.uc_stack.ss_size = STACK_SIZE;
thread_ctx.uc_stack.ss_flags = 0;
```

Listing 14. coroutine_demo.c(set up stack space)

```
swapcontext(&main_ctx, &thread_ctx);

for (int k = 0; k < 2; k++) {
    printf("[Main] Resumed from Thread (Round %d)
        .\n", k+1);
    printf("[Main] Main doing some work...\n");

    printf("[Main] Switching back to Thread...\n"
        );
    swapcontext(&main_ctx, &thread_ctx);
}
```

Listing 15. coroutine_demo.c(loop to execute coroutines)

C. Result Analysis

According to the running screenshot, 5 coroutines (Co 0 to Co 4) were created and initialized after the program started. During the execution process, the coroutines execute in a polling manner: first, each coroutine executes the first iteration, then the second iteration, and finally the third iteration. Coroutine 4 only requires 2 iterations to complete, making it the fastest coroutine to execute; And coroutine 3 requires 5 iterations to complete, making it the longest running coroutine. As each coroutine is completed, the system will update the number of active coroutines. When all coroutines are completed, the program returns to the main program and ends.

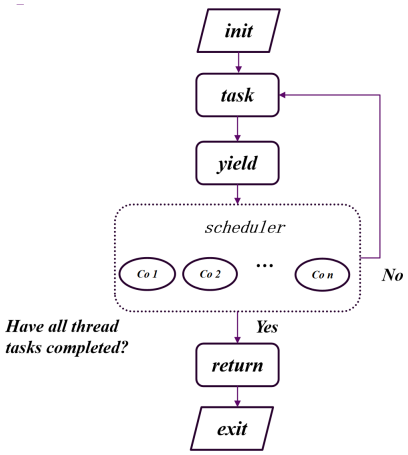

Fig. 1. Coroutines Workflow

The Gantt chart clearly displays the allocation of execution time slices for the coroutine. There are a total of 16 time slots, and the coroutine is executed sequentially according to the polling scheduling strategy. From the graph, it can be seen that coroutines 0, 1, and 2 each occupy 3 time slots, coroutine 4 occupies 2 time slots, and coroutine 3 occupies 5 time slots. This scheduling method ensures that each coroutine has the opportunity to execute, while actively relinquishing control through coroutines, achieving collaborative scheduling.

Through this experiment, a user state coroutine based on the ' ucontext. h ' library was successfully implemented, verifying its lightweight context switching advantage. Collaborative scheduling is achieved by actively calling swapcontext() to relinquish control, making it suitable for handling I/O-intensive tasks. Compared with kernel threads, the context switching of user state coroutines does not need to fall into kernel state, significantly reducing overhead. At the same time, by sharing memory space, the overhead of data transmission is reduced.

In the experiment, using a global array as stack space carries the risk of stack overflow. In practical projects, 'malloc()' should be used to dynamically allocate stack space. In terms of scheduling strategies, more complex scheduling algorithms can be implemented, such as priority based scheduling or time slice rotation scheduling, to adapt to the needs of different scenarios. In addition, adding a comprehensive error handling mechanism, implementing coroutine pooling to avoid the overhead of frequent creation and destruction of coroutines, and fully utilizing hardware resources by combining kernel threads in multi-core systems are all directions worth further exploration.

In summary, as a user mode concurrent programming model, coroutines provide an efficient solution for I/O-intensive tasks. This experiment successfully implemented the basic functions of user state coroutines, including context switching, stack space management, and simple scheduling. By analyzing the

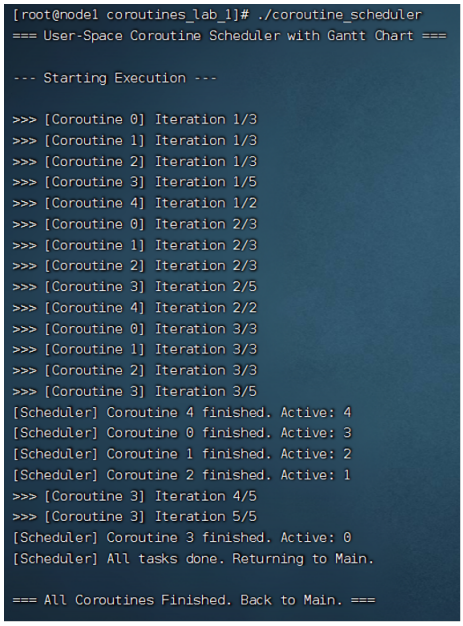

Fig. 2. Coroutines Result

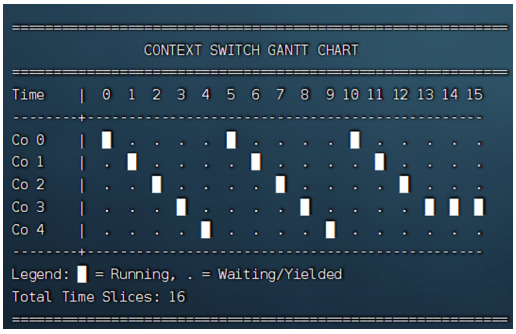

Fig. 3. Coroutines Result Gantt

running results, the advantages of coroutines in concurrent execution were verified, especially their lightweight context switching feature.

VIII. Conclusion

This paper presented a five-scenario quantitative experiment comparing kernel-level threads (KLT via Linux `kthread`) and user-level threads (ULT via POSIX pthreads) on an ARM64 Linux 6.8 system. Three principal conclusions emerge:

First, the Linux kernel's classification of context switches is determined strictly by the thread's state at the moment of the switch, not by the programmer's intent. `sched_yield()` and `schedule()` leave threads in `TASK_RUNNING` and therefore produce zero voluntary context switches (vol_csw $= 0$), despite appearing cooperative. `usleep(1)` and `usleep_range(1,2)` transition threads to `TASK_INTERRUPTIBLE` and produce exactly the expected ≈ 100 voluntary context switches per thread per run, confirmed with sub-unit precision across all five repetitions.

Second, more CPUs do not always improve throughput. When four threads compete for a single shared atomic counter, consolidating them on one CPU yields a $2.9\times$ (ULT) to $3.2\times$ (KLT) throughput improvement because the cache line holding the counter remains permanently in the `Modified` state in L1, eliminating the 100–300-cycle MESI invalidation round trips incurred under multi-CPU contention.

Third, KLT outperforms ULT by 25.4% at baseline and by up to 82.2% in yield-intensive scenarios, due to the elimination of user-to-kernel mode transitions during the hot loop and direct `task_struct` field access for context-switch accounting. The advantage vanishes in sleep-dominated scenarios where both models share the same hrtimer kernel path.

In addition, the advanced option topic on coroutines demonstrated multiple key technical points for implementing coroutines. Using the POSIX ‘ ucontext. h ‘ library as the implementation solution, its API design is clear, easy to understand and use, providing a reliable foundation for the implementation of coroutines. By separating the steps of context initialization, stack space allocation, and execution function association in the code structure, modular design has been achieved, improving the maintainability of the code. In terms of scheduling strategy, the experiment used polling scheduling to achieve simple and effective coroutine scheduling, laying the foundation for the development of more complex scheduling algorithms in the future. In terms of performance, experimental results show that coroutines can effectively utilize system resources during concurrent execution, especially exhibiting good responsiveness when handling multiple tasks.

In terms of engineering optimization, although using a global array as the stack space in the experiment simplifies implementation, dynamic memory allocation should be adopted in practical engineering, combined with mechanisms such as stack overflow detection, to improve system reliability. In terms of application scenarios, coroutines are particularly suitable for network programming, asynchronous I/O, and other scenarios, which can significantly improve the system's concurrent processing capability and response speed.

These findings provide code-level, mechanism-grounded explanations for performance differences that are often described only qualitatively in operating systems textbooks, demonstrating the value of hardware-grounded quantitative experiment design in OS education and system software development.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," ACM Transactions on Computer Systems, 1992.
[2] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley, 2010.
[3] F. J. Corbató et al., "Introduction and Overview of the Multics System," *Proceedings of the 1962 Fall Joint Computer Conference*, pp. 185-196, 1962.
[4] Arm Ltd., *ARM Architecture Reference Manual for A-profile architecture*. Arm DDI 0487E.a, 2023.
[5] P. E. McKenney, "Is Parallel Programming Hard, And, If So, What Can Be Done About It?" *The Linux Kernel Documentation*, 2017.