Towards Hybrid Cooperative-Preemptive Scheduling

Yizheng Xie
yizheng_xie@brown.edu
Brown University
Providence, USA

Di Jin
di_jin@brown.edu
Brown University
Providence, USA

Nikos Vasilakis
nikos@vasilak.is
Brown University
Providence, USA

Abstract

Cooperative scheduling avoids the many shared-state pitfalls of preemption, but risks fairness—in the limit resulting in denial of service and resource exhaustion. This paper argues that a careful hybrid between cooperation and preemption is *both* feasible and advantageous: by allowing only carefully controlled and developer-configurable preemption in an otherwise cooperative environment, the scheduler can maintain key invariants while restoring fairness. The paper presents a series of case-study workloads that motivate the need for preemption in real-world cooperative environments, sketches a hybrid design that introduces controlled preemption while maintaining cooperation benefits, and discusses the benefits by applying this hybrid design on the case-study workloads. A hybrid scheduling implementation, *Cx*, is in progress.

CCS Concepts: • **Software and its engineering → Scheduling**; **Coroutines**; **Concurrent programming structures**.

Keywords: Scheduling, Preemption, Cooperation, Event, OS Thread, Language Runtime, Concurrency

1 Introduction

Cooperative, or non-preemptive, scheduling is a common multitasking approach in which tasks yield voluntarily to one another—without being forced to do so by an external scheduler. Cooperation offers multiple benefits: it simplifies the management of state shared among tasks, which yield only after safely tidying up shared state, and improves performance, as tasks yield to each other directly—without the involvement of external scheduling, complex state save-restore, and other overheads. As a result, cooperative scheduling is used pervasively in numerous runtime environments—*e.g.,* JavaScript's `async`/`await` [45], Lua's coroutines [5], Julia's `Tasks` [2], and Rust's Tokio runtime [2, 5, 15].

Unfortunately, cooperative scheduling can allow a single task to monopolize execution and resources, either accidentally or intentionally. Since tasks yield control only voluntarily, any single task may decide to never yield—thus leading to delays in execution for the rest of the tasks, lack of fairness, accidental resource exhaustion, or even malicious attacks. Examples of such adverse effects due to resource monopoly in real-world environments include head-of-line blocking [19, 32, 42], denial-of-service (DoS) attacks [22, 38, 49], resource exhaustion [36], delayed garbage collection [18], and instability in constrained operating system implementations [1, 60].

This antithesis between cooperative and preemptive scheduling underpins a classical debate in the literature, including the controversy between (cooperatively scheduled) event-based concurrency constructs versus (preemptively scheduled) thread-based concurrency [17, 26, 27, 39, 55]. This and other heated debates in languages and systems [41, 48] are more about antithetical scheduling approaches than contrasting concurrency abstractions (§2).

The key thesis underlying this paper is that a carefully hybridized approach between cooperation and preemption is both *feasible* and *advantageous*. Feasibility stems from introducing carefully controlled, coarse-grained preemption into an otherwise cooperatively scheduled environment; and advantages stem from sane(r) management of shared state without the risk of execution monopoly.

The proposed hybrid approach, termed cooperative-with-controlled-preemption scheduling or *cocoon* scheduling, leverages the abstractions naturally found in cooperative environments. Introducing cocoon scheduling to applications requires importing a native library that introduces a set of abstractions and runtime support for careful preemption. Cocoon runtime combines controllable language-level interrupts with scheduling handlers invoked directly upon interrupt. Scheduling handlers can preempt running tasks and execute developer-defined preemption tasks that are carefully constrained to maintain key invariants. Preemption tasks are written in a subset of the source language designed to limit effects to shared state—ensuring safe and re-entrant execution. Developer-provided tasks written in this subset are checkable via ahead-of-time static program analysis that confirms the safety of these fragments before development, informing developers of any potential violations—*e.g.,* accidentally modifying shared state.

Apart from addressing inherent limitations in cooperative scheduling, cocoon additionally allows developers to interpose upon, control, and manipulate the task queues and related data structures of a cooperative scheduling environment. While this manipulation is constrained to ensure

PLOS '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2225-7/25/10
https://doi.org/10.1145/3764860.3768329

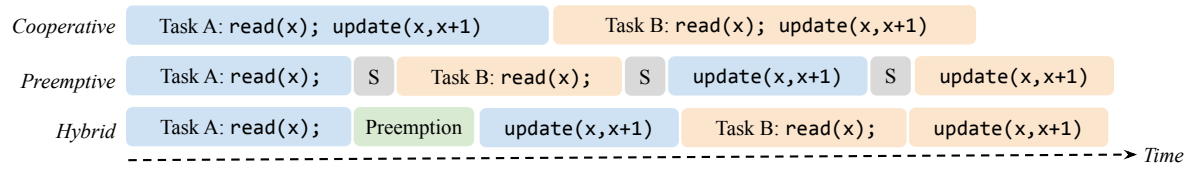

Figure 1. Different scheduling approaches. Two tasks A (blue) and B (orange) execute concurrently. Cooperative environments (top) run tasks to completion unless they explicitly yield. Preemptive environments (mid) invoke an external scheduler (gray), which switches tasks at predefined and, from the task perspective arbitrary, intervals—thus requiring synchronization. Cocoon scheduling allows for controlled, lightweight preemption of tasks (green) while preserving the semantics of the original program.

cocoon maintains safety, it is still powerful enough to satisfy the unmet needs of several practical scenarios (§3).

Starting with a discussion of prior scheduling work (§2), this paper introduces a series of scenarios that highlight fundamental scheduling limitations (§3), then introduces (§4) and applies (§5) cocoon scheduling to these scenarios, before concluding with broader possibilities (§6).

2 Background & Prior Hybrid Approaches

This section dives deeper into the various scheduling approaches, with key trade-offs summarized in Tab. 1, ending with relevant work in other hybrid approaches.

2.1 Cooperative scheduling

Historically, cooperative scheduling has been the first approach taken by the implementors of a multitasking environment as it is significantly easier to implement and reason about [29]: tasks are scheduled to run until they complete, unless they themselves voluntarily yield control. Consider the two tasks A and B at the top of Fig. 1, which read and update a global variable `x` shared between the two. Unless task A explicitly yields control by *e.g.,* invoking an `yield` or calling asynchronous operations (which it does not), task B will run only after A completes, *i.e.,* there is no way for B to observe inconsistent state. Even if another task—intended to preempt other tasks—arrives, it must wait until task A finishes.

Strengths. Cooperative multitasking reduces the risk of concurrency bugs caused by race conditions and misused synchronization primitives such as locks, as tasks yield only at explicitly defined points, allowing developers to tidy up shared state. Aside from its implementation simplicity, task switches are typically lightweight because they occur entirely in user space via explicit calls, without kernel involvement; the runtime saves minimal continuation state and resumes the task when it is runnable.

Due to its correctness, attractive simplicity, and performance characteristics, cooperative scheduling is common since the early days of operating systems, *e.g.,* the classic MacOS and Windows 1.x-3.11 in the nineties. It is widely used in modern programming languages runtimes and real-time operating systems. Programming abstractions such as *futures* and *promises* allow developers to express concurrency explicitly by `await`ing results [7, 11, 14], while asynchronous *callbacks* allow registering handlers to be invoked as responses to events such as incoming request or completion of disk I/O. Cooperative user-level threads are also prevalent in a wide range of runtime environments such as coroutines in Lua and C++, green threads in Julia, and fibers in Ruby and the Win32 API. Embedded and real-time systems such as Free RTOS [31] and RISC OS [1] leverage cooperative scheduling for efficiency.

Table 1. Comparison of cooperative and preemptive scheduling.

	Cooperative	**Preemptive**
Synchronization	Simple (yield)	Complex (many)
Fairness guarantee	Limited	Strong
Liveness guarantee	Limited	Strong
Scheduling granularity	Coarse	Fine
Context switch overhead	Low	High
Customizability	High	Low

Challenges. Unfortunately, as tasks themselves decide when to yield, they risk consuming unbounded execution resources, resulting in accidental or voluntary starvation. These risks are acutely exacerbated by modern trends: (1) the prevalence of complex software dependencies, including hundreds of transitive third-party libraries in modern applications [54], which might contain bugs or vulnerabilities monopolizing resources; (2) the use of large-language models for code generation by developers of varying skill and care [35]. Real cases of cooperative tasks monopolizing resources (§3) have rendered entire systems unusable [36, 49, 50].

2.2 Preemptive Scheduling

Preemptive scheduling typically leverages some form of external stimulus such as an interrupt or an OS signal to invoke a scheduler; In turn, the scheduler decides which task to run next according to some criteria, *e.g.,* fair sharing or priorities. Fig. 1 (middle) shows the scheduler (gray box) preempting Task A before its completion and scheduling Task B.

Strengths. Periodic and frequent such preemptions have multiple benefits. They ensure fair resource sharing by multiplexing tasks of various lengths and priorities, and improve

reliability by preventing excessive resource consumption—by suspending or terminating misbehaving tasks without disrupting others.

Unix and Unix-based operating systems (but not classic MacOS in the nineties) provided process-level preemptive multitasking out of the box—part of Unix's broader approach to user multiplexing. Modern systems such as Linux, the entire Windows NT family (including Windows 2000 and later), and mobile platforms like Android and iOS, also rely on kernel-level preemptive multitasking. Finer-grained abstractions such as POSIX and NPTL threads support kernel-level preemptive multitasking by allowing the kernel scheduler to make context-switching decisions independently of task cooperation. Beyond OS-level threads commonly exposed across programming languages, other preemptive abstractions are prevalent at the language level by user-level schedulers preempting and scheduling tasks. Examples include processes in Erlang [51], thread pools in C#, the ExecutorService in Java [6], and the Rayon parallelism library in Rust [12].

Challenges. But such preemption, as Dijkstra put it, "opens up the box of Pandora" [30]. As Fig. 1's task A is preempted before completion, task B updates a shared variable `x` with a stale value—which is preventable with mutex locks and other synchronization primitives, which in turn give rise to deadlocks, livelocks, and other pathologies that haunt developers. Entire generation of techniques has been developed to deal with these pathologies.

2.3 Hybrid scheduling

Prior work. The conflicting strengths of these two scheduling approaches have given rise to significant discussion in several communities. This includes classic debates such as threads *vs.* events [39], efforts introducing automatic stack management—*i.e.,* avoiding the need for callbacks—into cooperative task environments [17], decoupling thread representation from scheduler implementation [41]. These approaches mitigate the drawbacks of choosing either extreme, yet they do not integrate their complementary strengths.

Other hybrid scheduling approaches aim to combine specific benefits—*e.g.,* improving concurrency programming [24, 44, 58], ensuring fairness [33], or unifying abstractions [20, 23]. However, their approach and philosophy is different from cocoon scheduling, as they do not seek to strike a general and configurable sweet spot between preemptive and cooperative approaches.

A new approach. Cocoon scheduling prioritizes the key advantages of cooperative scheduling—such as the fully cooperative Node.js or Lua runtime—while incorporating a limited, customizable form of preemption. It is agnostic to the interrupt mechanism, a choice that plays a significant role in the resulting performance under certain workloads—multiple, possibly combined, choices would be possible. For example, recent user-level preemptive scheduling solutions rely on compiler instrumentation [34] or hardware-supported features such as posted interrupts [37] and user-level interrupts [32, 42]. Timeout-based preemption such as timed functions [22] and execution budgets [4] yield control to the scheduler at predefined timeouts, allowing the system to preempt tasks when they exceed their time budget.

3 Real-World Examples

This section presents three real-world scenarios that would benefit from cocoon scheduling (§4): mitigating DoS attacks, avoiding head-of-line blocking, and enabling timely overload control.

3.1 Regular Expression Denial-of-Service

Regular-Expression Denial of Service (ReDoS) is a class of denial-of-service attacks that exploit the exponential-time worst-case behavior of certain regular expression patterns, often rendering popular web servers unresponsive [36, 49, 50]. Such attacks induce excessive CPU usage by triggering catastrophic backtracking in regular expression engines. For example, poorly constructed pattern `(a+)+b` with nested quantifiers causes an exponential time complexity on input with repeated "a", e.g., `aaaaaaac`. As shown in the codes below, an evil input can make the server consume excessive CPU resources for infinite time in ln.3.

```
1  const server = http.createServer((req, res) => {
2      const regex = /(a+)+b/;
3      const match = regex.match(req.body);
4      res.send(match); });
```

This vulnerability can prevent other tasks from executing and monopolize resources, thus render applications unresponsive in cooperative environments [22, 38, 53].

3.2 Head-of-line Blocking

Head-of-line blocking, where a long-running task delays the execution of subsequent tasks, leads to high tail latency in server applications that handle heterogeneous workloads with dispersed request times—a common pattern in datacenter environments such as database systems and search engines [13, 21, 47]. For example, short-running GET requests (ln.1 below) to a database service may suffer elevated tail latency and reduced throughput when a long-running SCAN request (ln.5) blocks the queue.

```
1  dbserver.get('/get', (req, res) => {
2      const parsedUrl = url.parse(req.url, true);
3      const result = db.getById(parsedUrl.query.id);
4      res.send(result); });
5  dbserver.post('/scan', (req, res) => {
6      const query = url.parse(req.url, true).query;
7      // long-running scan operation
8      const result = db.scan(query.start, query.end);
9      res.send(result); });
```

Different request types (e.g., latency-critical vs. best-effort) or workload characteristics (e.g., traffic distributions) may also require customized scheduling priorities or granularities, informed by dynamic application needs or runtime metrics. For instance, heavy-tailed distributions such as bimodal workloads can benefit from shorter preemption quanta, reducing blocking in a timely manner [42].

3.3 Delayed Overload Control

Service overload, where traffic surges exceed service capacity, can severely degrade performance and responsiveness. For example, complex microservice architectures, widely adopted across various enterprises [52, 57], are particularly vulnerable to overload, such as demand spikes during major sales events. Effective overload control leverages diverse runtime metrics for early overload detection, and reacts to overload conditions, including distributed rate limiting [56, 59], granting credits to clients [25], and selective API throttling [46]. For example, the following overload control mechanism tracks request-queuing time—*i.e.,* the time between arrival and processing start (ln.2 below)—and propagates overload signals (ln.4) to coordinate adaptive rate limiting [59].

```
1  function handler(req, res) {
2    const queuingTime = Date.now() - req.arrivalTime;
3    if (queuingTime > THRESHOLD) {
4      propagateOverloadSignal(); }
5    processRequest(req, res); }
```

Such overload control—similar to other high-priority tasks, e.g., garbage collection or health checks [18]—requires timely execution to rapidly adapt to fluctuating workloads; however, in fully cooperative environments, a long-running task can delay overload detection and mitigation, exacerbating performance degradation.

4 Cocoon Scheduling

This section sketches the key elements of cocoon scheduling (Fig. 2), at times using examples from the Node.js internals, to enable safe and low-overhead user-level preemption without compromising the simplicity and efficiency of cooperative scheduling approaches.

4.1 Abstraction

Cocoon starts from a cooperative programming abstraction and introduces controlled, safe, and developer-configurable preemption at different levels of granularity, such as time-based or event-driven preemption. As illustrated in Fig. 1, cocoon scheduling preserves the *task* abstraction in cooperative scheduling, where a preemption task (green box)—a developer-defined specialized task—can preempt currently running tasks. It allows developers to define a scheduling *policy* that determines when the preemption happens and what the preemption task does.

Generalization and invariants. The cocoon abstraction is generalizable to different cooperative environments, but must preserve several key invariants of state management to ensure the safety of the preemption. For example, the preemption tasks can read the states of other tasks—to take them into account during scheduling decisions—but cannot modify them directly. Moreover, they can execute only at well-defined safe points—where program invariants are preserved—rather than at arbitrary points during execution. And the preempted tasks cannot access the preemption task's internal states—they remain preemption unaware.

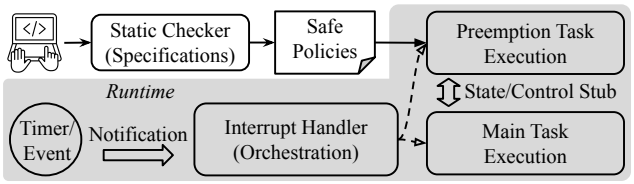

Figure 2. The design of a cocoon scheduler. Developer-provided scheduling policies, checked for their safety (left), are introduced to the runtime as handlers that run as a response to interrupts—*e.g.,* user interrupt from a dedicated timer core. Handlers cut into the execution of a cooperative task, switching to an isolated context to safely preempt without affecting the state of the original task.

4.2 Scheduling Policy

Policy registration. An example API snippet written in JavaScript illustrates how developers can define a policy to prevent infinite execution. The type (ln.2) of the policy indicates the preemption granularity, either time-based or event-driven preemption, *e.g.,* invoking the preemption task every 100 microseconds. The preemption task (ln.5) is expressed as a normal function using a *constrained subset* of the original language to perform controlled operations,*e.g.,* `throw` an exception to the preempted task or `continue` its execution.

```
1  const cocoon = require('cocoon-scheduler');
2  cocoon.registerCustomPolicy({
3    type: 'time', // time-based preemption
4    interval: 100, // us
5    task: (ctx) => {
6      if (cocoon.getCurrentTaskExecutionTime() > 1000) {
7        // if the current task exceeds 1s, throw an exception
8        cocoon.throw(new Error('Execution time exceeded'));
9      }
10     cocoon.continue(); // continue execution
11   }
12 })
```

To simplify policy registration, the scheduler can provide predefined policies to address common use cases. For example, a policy that preempts regular expression matches exceeding a specified time limit can be registered as follows:

```
1  // register a pre-defined policy
2  cocoon.registerTimeEnforcementPolicy({
3    func: 'regex.match', // function to check
4    timeLimit: 1000, // ms
5    task: (ctx) => cocoon.throw(
6      new Error('Execution time exceeded')
```

```
7     ),
8   })
```

Policies often prioritize certain tasks over others, e.g., prioritizing latency-critical tasks over best-effort ones. Given the aforementioned invariants and the potential risks illustrated in Fig. 1, the developer should explicitly register preemptible tasks and safe tasks to allow the scheduler to safely preempt them, e.g., preempting the SCAN handler (as preemptible tasks in ln.3) when the GET handler (as safe tasks in ln.4) is ready.

```
1  function ScanHandler(req, res) { ... }
2  function GetHandler(req, res) { ... }
3  cocoon.registerPreemptibleFunc(ScanHandler);
4  cocoon.registerSafeFunc(GetHandler);
```

Expressiveness. Preemption tasks can define custom states that are accessible exclusively to preemption tasks, e.g., a global map tracking request distribution. To allow communication between preemption tasks and the main task, the policy allows registering shared states that are writable by preemption tasks and only readable by the main task.

```
1  cocoon.registerState({ overloaded: false, })
```

Depending on the language runtimes, the preemption tasks can access main task's internal states and various runtime states related to its execution context, *e.g.,* the call stack, resource usage, and event queue length.

Policy update and cancellation. The scheduler can provide a set of control APIs that allow dynamically changing scheduling behavior at runtime, including updating and canceling registered policies. For example, a policy designed to mitigate head-of-line blocking may reduce the scheduling time quantum for aggressive preemption upon observing heavy-tailed workload distribution, or cancel the policy when preemption is no longer needed.

Ensuring safety. There are several constraints that policies must satisfy. First, policies must not import third-party libraries, as these may introduce side effects, external dependencies, or malicious codes that interfere with application execution. Second, policies must be deterministic and guaranteed to terminate, avoiding unbounded execution that could stall progress. Third, policies must not perform non-blocking operations (e.g., I/O or network calls) that complete asynchronously after the handler completes execution. Finally, policies cannot modify the application's internal state unless explicitly registered as shared states, preventing breaking language-level semantics and ensuring atomicity guarantees.

Developer efforts and trust. As with other cooperative scheduling systems, Cocoon offloads the responsibility of defining scheduling policies to developers. Unlike manual approaches that require inserting yield points at the statement level, however, Cocoon supports true preemption while minimizing developer burden: it provides lightweight preemptive capabilities that can be selectively and easily integrated into existing applications through provided APIs, as showcased in Section 5. While static analysis ensures aforementioned safety guarantees even when developers inadvertently introduce unsafe code changes, cocoon ultimately relies on developers to define and enforce benign application behaviors.

4.3 Mechanism

Fig. 2 illustrates the runtime architecture of a cocoon scheduler, which consists of three main components: policy registration, notification, and preemption execution. Language runtimes can selectively implement these mechanisms based on internal capabilities, minimizing preemption overhead while ensuring safety and efficiency.

Registration. Developers register preemptive policies using the registration API (solid arrows in Fig. 2). Upon registration, the runtime performs static analysis to verify the safety of the policy, ensuring that it adheres to aforementioned invariants and constraints. If verification succeeds, the scheduler initializes a sandbox or an isolated context (e.g., a separate `v8::Isolate` in the Node.js runtime) to safely evaluate the policy. This isolation preserves the semantics of the underlying execution abstraction and prevents corruption of currently running task's runtime context.

Notification. The runtime supports multiple notification mechanisms to trigger preemption, including timer-based and event-driven notifications. For timer-based preemption, the runtime sets up a dedicated timer process that notifies the Node.js runtime at regular intervals via general-purpose OS signals or low-overhead, hardware-supported interrupts. For example, user-level interprocessor interrupts (UIPI) reduce the preemption overhead from 2.4 µs (for signal handling) to 0.4 µs [32, 42]. To handle notifications, the runtime registers a handler (e.g., signal handler) to trigger preemption. For event-driven preemption, the runtime hooks into the underlying asynchronous interface (e.g., the `libuv` event loop in Node.js) to trigger preemption when new events are enqueued, such as incoming requests or completed I/O operations.

Preemption. When a notification arrives, the runtime interrupts the main execution at a safe point where program state is well-defined and consistent, e.g., via V8's `RequestInterrupt` API in Node.js. The runtime then switches to the isolated execution context to execute the registered preemption task (dashed arrows in Fig. 2). This isolated context maintains its own execution state and operates on preemption-specific states, separate from the main task's execution context. Upon completion of the preemption task, the runtime determines switching back to the main task's execution context. The runtime applies the selected action before switching back to the main task's execution context.

Stubs. The runtime implements state and control stubs that bridge the isolated preemption execution context with the main task's execution context. State stubs allow preemption tasks to inspect main task's internal states and execution context without directly corrupting its memory. These

stubs can also update global variables explicitly registered as shared state. Control stubs perform operations that affect main task's behavior according to execution of preemption tasks, such as scheduling an exception to the main task's context using V8's `RequestInterrupt` API. They also support dynamic modifications to registered policies, including adjusting the preemption quantum or canceling a policy.

5 Applying Cocoon Scheduling

This section revisits aforementioned examples (§3) and illustrates how developers can leverage cocoon scheduling (§4) to address their problems.

5.1 ReDoS Prevention

Cocoon scheduling enables developers to enforce fine-grained preemption policies on regular expression operations and other long-running tasks by registering a default policy that preempts regex matches exceeding a specified time limit (e.g., 100 ms), as illustrated in §4. To support this, the scheduler spawns a dedicated timer process that periodically notifies the Node.js runtime—e.g., via low-level user interrupts—to inspect the call stack and track cumulative execution time in target functions such as `regex.match`. If a regex exceeds its assigned time budget, the scheduler safely interrupts the task and throws an exception into the corresponding V8 isolate. This exception can then be caught and handled by the application using the following code snippet, preventing ReDoS attacks and preserving responsiveness:

```
// create preemptive regex server
const server = http.createServer((req, res) => {
    const regex = /^([a-zA-Z]+)+$/;
    try {
        const match = regex.match(req.body);
        res.send(match);
    } catch (err) {
        if (err instanceof cocoon.PreemptedError) {
            res.status(500).send('Interrupted');
        }
    }
});
```

5.2 Head-of-line Blocking Mitigation

As illustrated in the code snippet below, cocoon scheduling allows developers to register a long-running `server.scan` function as preemptible, a short-running `server.get` function as safe, and a policy at event-based granularity (or a specific time quanta such as 5µs) that preempts `server.scan` when a new request arrives.

```
cocoon.registerPreemptibleFunc(server.scan);
cocoon.registerSafeFunc(server.get);
cocoon.registerDefaultPreemptionPolicy({
  type: 'event', // event-based preemption
})
```

By explicitly registering preemptible and safe functions, the scheduler preserves the atomicity guarantees of cooperative scheduling while enabling selective preemption. The scheduler supports preemption at event granularity—e.g., extending the Node.js runtime—to avoid the overhead of coarse-grained time-based preemption when no events are pending. When a new `get` request arrives, the scheduler interrupts the main application isolate if the currently executing task is `server.scan`. It switches to a dedicated V8 isolate to handle the get request without corrupting application state. After the handler completes, the scheduler resumes execution of the previously preempted task.

5.3 Overload Control

Cocoon scheduling enables developers to inspect various runtime states at arbitrary points in time, including call stacks, resource usage, and event queues. It also supports programmable and timely reactions to profiling events. As shown in the code snippet below, developers can register a safe handler that executes specific actions upon overload. They can then define a scheduling policy that periodically monitors the incoming request queue and invokes the handler whenever the queue length exceeds a predefined threshold. In this way, cocoon scheduling enables overload control mechanisms that are both informed and timely—by exposing runtime states and allowing preemption.

```
cocoon.registerState({ overloaded: false, })
cocoon.registerSafeFunc(OverloadAction);
cocoon.registerCustomPolicy({
  type: "time",
  interval: 100, // microseconds
  task: (ctx) => {
    if (cocoon.getQueueLength() > threshold) {
      OverloadAction();
    }
    cocoon.continue(); // continue execution
  }
})
```

6 Broader Horizons

As shown earlier, with careful design cocoon scheduling is feasible (§4) and has the potential to aleviate key limitations of cooperative scheduling (§5) without opening up Pandora's Box [30]. Once available, it would also enable further opportunities across several other areas that go beyond direct scheduling concerns.

Observability and profiling. Profiling tools like distributed tracing [9, 10] and observability frameworks such as service meshes [3, 8] typically rely on coarse-grained, external metrics such as service dependencies and resource usage. Without further instrumentation, these tools often cannot distinguish whether high latency is caused by resource contention between tasks or by a single task monopolizing the CPU. Cocoon scheduling might enable more accurate diagnosis of performance issues by introducing lightweight, low-overhead, and reactive profiling policies that monitor transient states and resource usage at a finer granularity, *e.g.*,

per-handler CPU time, memory footprint, or request queue length.

Timing-related security. Systems often require both strong timing guarantees and isolation to prevent interference between components when enforcing timing constraints [43]. Other environments, such as serverless platformss often rely on lightweight isolation that does not use stronger and preemptive isolation such as containers or VMs [16]. Cocoon scheduling can enhance the security of these environments by allowing such security policies—e.g., timing, resources, and access control—to execute in isolated contexts.

Memory Management. Memory management significantly impacts data locality, memory access efficiency, and garbage collection performance. For example, cocoon scheduling can mitigate inter-task cache interference caused by arbitrary preemptions [40], thereby improving memory efficiency while still preserving preemption capabilities. Also, cocoon scheduling can potentially enable efficient garbage collection by proactively monitoring memory pressure and CPU idle time, enabling timely and minimally disruptive garbage collection cycles [18, 28].

Acknowledgments

We are thankful to anonymous PLOS reviewers. This material is based upon research supported by NSF awards CNS-2247687 and CNS-2312346, DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a seed grant from Brown University's Data Science Institute, and a BrownCS Faculty Innovation Award.

References

[1] 2024. RISC OS. https://www.riscos.info/index.php/RISC_OS. Accessed: June 2025.
[2] 2025. Asynchronous Programming. https://docs.julialang.org/en/v1/manual/asynchronous-programming/#man-asynchronous. Accessed: June 2025.
[3] 2025. Cilium. https://cilium.io/. Accessed: June 2025.
[4] 2025. Consume Budget. https://docs.rs/tokio/latest/tokio/task/coop/fn.consume_budget.html. Accessed: June 2025.
[5] 2025. Coroutines. https://www.lua.org/pil/9.1.html. Accessed: June 2025.
[6] 2025. ExecutorService. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html. Accessed: June 2025.
[7] 2025. Futures and async syntax. https://doc.rust-lang.org/book/ch17-01-futures-and-syntax.html. Accessed: June 2025.
[8] 2025. Istio. https://istio.io/. Accessed: June 2025.
[9] 2025. Jaeger. https://www.jaegertracing.io/. Accessed: June 2025.
[10] 2025. OpenTelemetry. https://opentelemetry.io/. Accessed: June 2025.
[11] 2025. Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed: June 2025.
[12] 2025. Rayon. https://docs.rs/rayon/latest/rayon/. Accessed: June 2025.
[13] 2025. RocksDB. https://rocksdb.org/. Accessed: June 2025.
[14] 2025. std::promise. https://cplusplus.com/reference/future/promise/. Accessed: June 2025.
[15] 2025. tokio. https://tokio.rs. Accessed: June 2025.
[16] Zack Bloom. 2018. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers/. Accessed: June 2025.
[17] Atul Adya, Jon Howell, Marvin Theimer, Bill Bolosky, and John Douceur. 2002. Cooperative task management without manual stack management. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*.
[18] Austin Clements. 2019. Non-Cooperative Preemption. https://go.googlesource.com/proposal/+/master/design/24543-non-cooperative-preemption.md. Accessed: June 2025.
[19] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. 2025. Extended User Interrupts (xUI): Fast and Flexible Notification without Polling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 373–389.
[20] Guy E Blelloch, Lenore Blum, Mor Harchol-Balter, and Robert Harper. 2020. Multiscale Scheduling: Integrating Competitive and Cooperative Scheduling in Theory and in Practice. (2020).
[21] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the" micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 645–650.
[22] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2020. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 465–477.
[23] Frédéric Boussinot. 2006. FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience* 18, 5 (2006), 445–469.
[24] Pavol Černỳ, Edmund M Clarke, Thomas A Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. 2017. From non-preemptive to preemptive scheduling using synchronization synthesis. *Formal methods in system design* 50, 2 (2017), 97–139.
[25] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload control for {μs-scale}{RPCs} with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 299–314.
[26] Ryan Cunningham and Eddie Kohler. 2005. Making Events Less Slippery with eel.. In *HotOS*.
[27] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, and Robert Morris. 2002. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. 186–189.
[28] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. 2016. Idle time garbage collection scheduling. *ACM SIGPLAN Notices* 51, 6 (2016), 570–583.
[29] Edsger W. Dijkstra. 1967. The structure of the "THE"-multiprogramming system. In *Proceedings of the First ACM Symposium on Operating System Principles (SOSP '67)*. Association for Computing Machinery, New York, NY, USA, 10.1–10.6. https://doi.org/10.1145/800001.811672
[30] Edsger W. Dijkstra. 2000. EWD1303: My recollections of operating system design. https://www.cs.utexas.edu/~EWD/transcriptions/EWD13xx/EWD1303.html. Accessed: June 2025.
[31] FreeRTOS. 2025. Tasks and Co-routines. https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/01-Tasks-and-co-routines/00-Tasks-and-co-routines.
[32] Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout. 2025. The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 1015–1032.
[33] Reiner Hähnle and Ludovic Henrio. 2023. Provably fair cooperative scheduling. *arXiv preprint arXiv:2312.16977* (2023).
[34] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 466–481.

[35] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).

[36] John Graham-Cumming. 2019. Details of the Cloudflare Outage on July 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/. Accessed: June 2025.

[37] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[38] Kenton Varda. 2018. Webassembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/. Accessed: June 2025.

[39] Hugh C Lauer and Roger M Needham. 1979. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review* 13, 2 (1979), 3–19.

[40] Sheayun Lee, Sang Lyul Min, Chong Sang Kim, Chang-Gun Lee, and Minsuk Lee. 1999. Cache-conscious limited preemptive scheduling. *Real-Time Systems* 17, 2 (1999), 257–282.

[41] Peng Li and S Zdancewic. 2006. A language-based approach to unifying events and threads (2006).

[42] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. 2024. Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–936.

[43] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2019. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–31.

[44] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. 2005. Preemptible atomic regions for real-time Java. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*. IEEE, 10–pp.

[45] OpenJS Foundation. 2025. Node.js. https://nodejs.org/en. Accessed: June 2025.

[46] Jinwoo Park, Jaehyeong Park, Youngmok Jung, Hwijoon Lim, Hyunho Yeo, and Dongsu Han. 2024. TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 876–890.

[47] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 498–514.

[48] KC Sivaramakrishnan, Tim Harris, Simon Marlow, and Simon Peyton Jones. 2016. Composable scheduler activations for Haskell. *Journal of Functional Programming* 26 (2016), e9.

[49] Stack Overflow. 2016. Outage Postmortem - July 20, 2016. http://web.archive.org/web/20180801005940/http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016. Accessed: June 2025.

[50] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: a study of {ReDoS} vulnerabilities in {JavaScript-based} web servers. In *27th USENIX security symposium (USENIX Security 18)*. 361–376.

[51] Fredrik Stenmans. 2025. The Beam Book. https://blog.stenmans.org/theBeamBook/. Accessed: June 2025.

[52] Sudhir Tonse. 2015. Scalable Microservices at Netflix. Challenges and Tools of the Trade. https://www.infoq.com/presentations/netflix-ipc/

[53] C Joseph Vanderwaart. 2006. *Static enforcement of timing policies using code certification.* Technical Report.

[54] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2017. Towards Fine-grained, Automated Application Compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (Shanghai, China) *(PLOS '17)*. Association for Computing Machinery, New York, NY, USA, 43–50. https://doi.org/10.1145/3144555.3144563

[55] Rob Von Behren, Jeremy Condit, and Eric Brewer. 2003. Why events are a bad idea (for {High-Concurrency} servers). In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*.

[56] Jiali Xing, Akis Giannoukos, Paul Loh, Shuyue Wang, Justin Qiu, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C Lee. 2025. Rajomon: Decentralized and Coordinated Overload Control for {Latency-Sensitive} Microservices. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 21–36.

[57] Alex Xu. 2022. Twitter architecture 2022 vs. 2012. what's changed over the past 10 years? https://blog.bytebytego.com/p/twitter-architecture-2022-vs-2012

[58] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. 2011. Cooperative reasoning for preemptive execution. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. 147–156.

[59] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 149–161.

[60] Yousaf Bin Zikria, Sung Won Kim, Oliver Hahm, Muhammad Khalil Afzal, and Mohammed Y Aalsalem. 2019. Internet of Things (IoT) operating systems management: Opportunities, challenges, and solution. *Sensors* 19, 8 (2019), 1793.