Lightweight Preemptible Functions

Sol Boucher, *Carnegie Mellon University;* Anuj Kalia, *Microsoft Research;* David G. Andersen, *Carnegie Mellon University;* Michael Kaminsky, *BrdgAI / Carnegie Mellon University*

https://www.usenix.org/conference/atc20/presentation/boucher

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Lightweight Preemptible Functions

Sol Boucher,* Anuj Kalia,† David G. Andersen,* and Michael Kaminsky‡*
*Carnegie Mellon University †Microsoft Research ‡BrdgAI

Abstract

Lamenting the lack of a natural userland abstraction for preemptive interruption and asynchronous cancellation, we propose lightweight preemptible functions, a mechanism for synchronously performing a function call with a precise timeout that is lightweight, efficient, and composable, all while being portable between programming languages. We present the design of *libinger*, a library that provides this abstraction, on top of which we build *libturquoise*, arguably the first general-purpose and backwards-compatible preemptive thread library implemented entirely in userland. Finally, we demonstrate this software stack's applicability to and performance on the problems of combatting head-of-line blocking and time-based DoS attacks.

1 Introduction

After years of struggling to gain adoption, the coroutine has finally become a mainstream abstraction for cooperatively scheduling function invocations. Languages as diverse as C#, JavaScript, Kotlin, Python, and Rust now support "async functions," each of which expresses its dependencies by "awaiting" a **future** (or promise); rather than polling, the language yields if the awaited result is not yet available.

Key to the popularity of this concurrency abstraction is the ease and seamlessness of parallelizing it. Underlying most futures runtimes is some form of green threading library, typically consisting of a scheduler that distributes work to a pool of OS-managed worker threads. Without uncommon kernel support (e.g., scheduler activations [3]), however, this logical threading model renders the operating system unaware of individual tasks, meaning context switches are purely cooperative. This limitation is common among userland thread libraries, and illustrates the need for a mechanism for *preemptive* scheduling at finer granularity than the kernel thread.

In this paper, we propose an abstraction for calling a function with a timeout: Once invoked, the function runs on the same thread as the caller. Should the function time out, it is preempted and its execution state is returned as a continuation in case the caller later wishes to resume it. The abstraction is exposed via a wrapper function reminiscent of a thread spawn interface such as `pthread_create()` (except *synchronous*). Despite their synchronous nature, **preemptible functions** are useful to programs that are parallel or rely on asynchronous I/O; indeed, we later demonstrate how our abstraction composes with futures and threads.

The central challenge of introducing preemption into the contemporary programming model is supporting existing code. Despite decades of improvement focused on thread safety, modern systems stacks still contain critical nonreentrancy, ranging from devices to the dynamic memory allocator's heap region. Under POSIX, code that interrupts other user code is safe only if it restricts itself to calling async-signal-safe (roughly, reentrant) functions [27]. This restriction is all too familiar to those programmers who have written signal handlers: it is what makes it notoriously difficult to write nontrivial ones. Preemption of a timed function constitutes its interruption by the rest of the program. This implies that *the rest of the program* should be restricted to calling reentrant functions; needless to say, such a rule would be crippling. Addressing this problem is one of the main contributions of this paper. Our main insight here, as shown in Figure 1, is that some libraries are naturally reentrant, while many others can be made reentrant by automatically cloning their internal state so that preempting one invocation does not leave the library "broken" for concurrent callers.

The most obvious approach to implementing preemptible functions is to map them to OS threads, where the function would run on a new thread that could be cancelled upon timeout. Unfortunately, cancelling a thread is also hard. UNIX's pthreads provide asynchronous cancelability, but according to the Linux documentation, it

> is rarely useful. Since the thread could be cancelled at *any* time, it cannot safely reserve resources (e.g., allocating memory with `malloc()`), acquire mutexes, semaphores, or locks, and so on... some internal data structures (e.g., the linked list of free blocks managed by the `malloc()` family of functions) may be left in an inconsistent state if cancellation occurs in the middle of the function call [25].

[†] This author was at Carnegie Mellon during this project.
[‡] This author was *not* at Carnegie Mellon during this project.

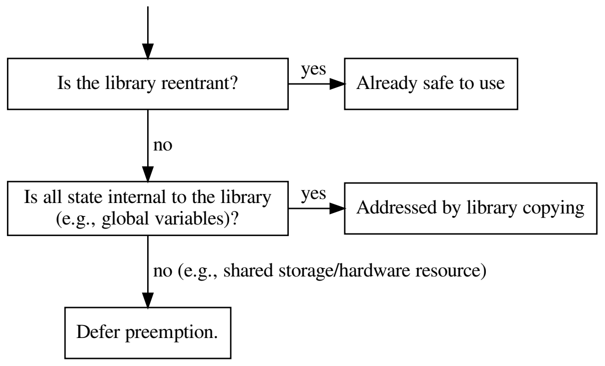

Is the library reentrant?
yes
Already safe to use
no
Is all state internal to the library (e.g., global variables)?
yes
Addressed by library copying
no (e.g., shared storage/hardware resource)
Defer preemption.

Figure 1: Taxonomy of support for library code. In practice, we always apply one of the two mitigations. Library copying is used by default, and is discussed in Sections 5.1 and 5.2. Deferred preemption is needed to preserve the semantics of `malloc()` and users of uncopyable resources such as file descriptors or network adapters, and is applied according to a whitelist, as described in Section 5.5.

The same is true on Windows, whose API documentation warns that asynchronously terminating a thread

> can result in the following problems: If the target thread owns a critical section, the critical section will not be released. If the target thread is allocating memory from the heap, the heap lock will not be released...

and goes on from there [28].

One might instead seek to implement preemptible functions via the UNIX `fork()` call. Assuming a satisfactory solution to the performance penalty of this approach, one significant challenge would be providing bidirectional object visibility and ownership. In a model where each timed function executes in its own child process, not only must data allocated by the parent be accessible to the child, but the opposite must be true as well. The fact that the child may terminate before the parent raises allocation lifetime questions. And all this is without addressing the difficulty of even calling `fork()` in a multithreaded program: because doing so effectively cancels all threads in the child process except the calling one, the child process can experience the same problems that plague thread cancellation [4].

These naïve designs share another shortcoming: in reducing preemptible functions to a problem of parallelism, they hurt performance by placing thread creation on the critical path. Thus, the state-of-the-art abstractions' high costs limit their composability. We observe that, when calling a function with a timeout, it is concurrency alone—and not parallelism—that is fundamental. Leveraging this key insight, we present a design that *separates interruption from asynchrony* in order to provide *preemption at granularities in the tens of microseconds*, orders of magnitude finer than contemporary OS schedulers' millisecond timescales. Our research prototype[1] is implemented entirely in userland, and requires neither custom compiler or runtime support nor managed runtime features such as garbage collection.

This paper makes three primary contributions: (1) It proposes function calls that return a continuation upon preemption, a novel primitive for unmanaged languages. (2) It introduces selective relinking, a compiler-agnostic approach to automatically lifting safety restrictions related to nonreentrancy. (3) It demonstrates how to support asynchronous function cancellation, a feature missing from state-of-the-art approaches to preemption, even those that operate at the coarser granularity of a kernel thread.

2 Related work

A number of past projects (Table 1) have sought to provide bounded-time execution of chunks of code at subprocess granularity. For the purpose of our discussion, we refer to a portion of the program whose execution should be bounded as **timed code** (a generalization of a preemptible function); exactly how such code is delineated depends on the system's interface.

Interface notwithstanding, the systems' most distinguishing characteristic is the mechanism by which they enforce execution bounds. At one end of the spectrum are **cooperative** multitasking systems where timed code voluntarily cedes the CPU to another task via a runtime check. (This is often done implicitly; a simple example is a compiler that injects a conditional branch at the beginning of any function call from timed code.) Occupying the other extreme are **preemptive** systems that externally pause timed code and transfer control to a scheduler routine (e.g., via an interrupt service routine or signal handler, possibly within the language's VM).

The cooperative approach tends to be unable to interrupt two classes of timed code: (1) **blocking-call** code sections that cause long-running kernel traps (e.g., by making I/O system calls), thereby preventing the interruption logic from being run; and (2) **excessively-tight loops** whose body does not contain any yield points (e.g., spin locks or long-running CPU instructions). Although some cooperative systems refine their approach with mechanisms to tolerate either blocking-call code sections [1] or excessively-tight loops [32], we are not aware of any that are capable of handling both cases.

One early instance of timed code support was the *engines* feature of the Scheme 84 language [15]. Its in-

[1]Our system is open source; the code is available from `efficient.github.io/#lpf`.

			Dependencies		Third-party code support	
System	Preemptive	Synchronous	In userland	Works without GC	Preemptible	Works without recompiling
Scheme engines	✓*	✓	✓		†	✓
Lilt		✓	✓		†*	—
goroutines			✓		†*	—
C∀	✓		✓	✓	†*	—
RT library	✓		✓	✓		✓
Shinjuku	✓			✓	†	
libinger	✓	✓	✓	✓	✓	✓

✓* = the language specification leaves the interaction with blocking system calls unclear
† = assuming the third-party library is written in a purely functional (stateless) fashion
†* = the third-party code must be written in the language without foreign dependencies
(beyond simple recompilation, this necessitates porting)

Table 1: Systems providing timed code at sub-process granularity

terface was a new `engine` keyword that behaved similarly to `lambda`, but created a special "thunk" accepting as an argument the number of ticks (abstract time units) it should run for. The caller also supplied a callback function to receive the timed code's return value upon successful completion. Like the rest of the Scheme language, engines were stateless: whenever one ran out of computation time, it would return a replacement engine recording the point of interruption. Engines' implementation relied heavily on Scheme's managed runtime, with ticks corresponding to virtual machine instructions and cleanup handled by the garbage collector. Although the paper mentions timer interrupts as an alternative, it does not evaluate such an approach.

Lilt [32] introduced a language for writing programs with statically-enforced timing policies. Its compiler tracks the possible duration of each path through a program and inserts yield operations wherever a timeout could possibly occur. Although this approach requires assigning the execution limit at compile time, the compiler is able to handle excessively-tight loops by instrumenting backward jumps. Blocking-call functions remained a challenge, however: handling them would have required operating system support, reminiscent of *Singularity*'s static language-based isolation [12].

Some recent languages offer explicit userland threading, which could be used to support timed code. One example is the Go language's [1] *goroutines*. Its runtime includes a cooperative scheduler that conditionally yields at function call sites. This causes problems with tight loops, which require the programmer to manually add calls to the `runtime.Gosched()` yield function [7].

The solutions described thus far all assume languages with a heavyweight, garbage-collected runtime. However, two recent systems seek to support timed code with fewer dependencies: the *C∀* language [8] and a C thread library for realtime systems (here, "*RT*") developed by Mollison and Anderson [22]. Both perform preemption using timer interrupts, as proposed in the early Scheme engines literature. They install a periodic signal handler for scheduling tasks and migrating them between cores, a lightweight approach that achieves competitive scheduling latencies. However, as explained later in this section, the compromise is interoperability with existing code.

Shinjuku [17] is an operating system designed to perform preemption at microsecond scale. Built on the Dune framework [5], it runs tasks on a worker thread pool controlled by a single centralized dispatcher thread. The latter polices how long each task has been running and sends an inter-processor interrupt (IPI) to any worker whose task has timed out. The authors study the cost of IPIs and the overheads imposed by performing them within a VT-x virtual machine, as required by Dune. They then implement optimizations to reduce these overheads at the expense of Shinjuku's isolation from the rest of the system.

As seen in Section 1, nonreentrant interfaces are incompatible with externally-imposed time limits. Because such interfaces are prolific in popular dependencies, no prior work allows timed code to transparently call into third-party libraries. Scheme engines and Lilt avoid this issue by only supporting functional code, which cannot have shared state. Go is able to preempt goroutines written in the language itself, but a goroutine that makes any foreign calls to other languages is treated as nonpreemptible by the runtime's scheduler [11]. The C∀ language's preemption model is only safe for functions guarded by its novel monitors: the authors caution that "any challenges that are not [a result of extending monitor semantics] are considered as solved problems and therefore not discussed." With its focus on realtime embedded systems, RT assumes that the timed code in its threads will avoid shared state; this assumption

```
struct linger_t {
    bool is_complete;
    cont_t continuation;
};

linger_t launch(Function func,
                u64 time_us,
                void *args);
void resume(linger_t *cont, u64 time_us);
```

Listing 1: Preemptible functions core interface

```
linger = launch(task, TIMEOUT, NULL);
if (!linger.is_complete) {
    // Save @linger to a task queue to
    // resume later
    task_queue.push(linger);
}

// Handle other tasks
...
// Resume @task at some later point
linger = task_queue.pop();
resume(&linger, TIMEOUT);
```

Listing 2: Preemptible function usage example

mostly precludes calls to third-party libraries, though the system supports the dynamic memory allocator by treating it as specifically nonpreemptible. Rather than dealing with shared state itself, Shinjuku asks application authors to annotate any code with potential concurrency concerns using a nonpreemptible `call_safe()` wrapper.

3 Timed functions: *libinger*

To address the literature's shortcomings, we have developed *libinger*,[2] a library providing a small API for timed function dispatch (Listing 1):

- `launch()` invokes an ordinary function `func` with a time cap of `time_us`. The call to `launch()` returns when `func` completes, or after approximately `time_us` microseconds if `func` has not returned by then. In the latter case, *libinger* returns an opaque continuation object recording the execution state.
- `resume()` causes a preemptible function to continue after a timeout. If execution again times out, `resume()` updates its continuation so the process may be repeated. Resuming a function that has already returned has no effect.

Listing 2 shows an example use of *libinger* in a task queue manager designed to prevent latency-critical tasks from blocking behind longer-running ones. The caller invokes a task with a timeout. If the task does not complete within the allotted time, the caller saves its continuation in the task queue, handles other tasks, and later resumes the first task.

[2]In the style of GNU's *libiberty*, we named our system for the command-line switch used to link against it. As the proverb goes, "Don't want your function calls to linger? Link with `-linger`."

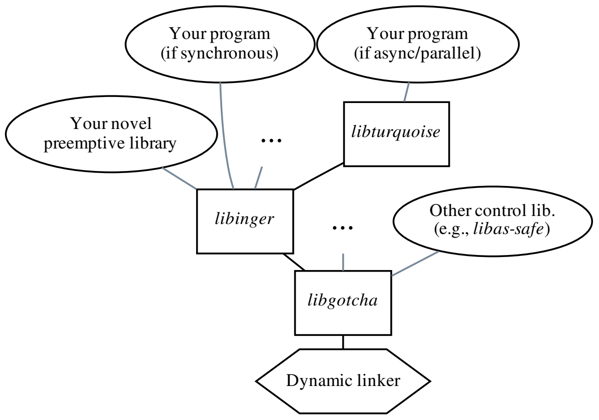

Your program (if synchronous)
Your program (if async/parallel)
Your novel preemptive library
...
libturquoise
libinger
...
Other control lib. (e.g., libas-safe)
libgotcha
Dynamic linker

Figure 2: Preemptible functions software stack. Hexagonal boxes show the required runtime environment. Rectangular boxes represent components implementing the preemptible functions abstraction. Ovals represent components built on top of these. A preemptible function's body (i.e., `func`) may be defined directly in your program, or in some other loaded library.

In accordance with our goal of language agnosticism, *libinger* exposes both C and Rust [2] APIs. To demonstrate the flexibility and composability of the preemptible function abstraction, we have also created *libturquoise*, a preemptive userland thread library for Rust, by porting an existing futures-based thread pool to *libinger*. We discuss this system in Section 4.

Figure 2 shows a dependency graph of the software components comprising the preemptible functions stack. The *libinger* library itself is implemented in approximately 2,500 lines of Rust. To support calls to nonreentrant functions, it depends on another library, *libgotcha*, which consists of another 3,000 lines of C, Rust, and x86-64 assembly. We cover the details in Section 5.

We now examine *libinger*, starting with shared state.

3.1 Automatic handling of shared state

As we found in Section 1, a key design challenge facing *libinger* is the shared state problem: Suppose a preemptible function F calls a stateful routine in a third-party library L, and that F times out and is preempted by *libinger*. Later, the user invokes another timed function F_0, which also calls a stateful routine in L. This pattern involves an unsynchronized concurrent access to L. To avoid introducing such bugs, *libinger* must hide state modifications in L by F from the execution of F_0.

One non-solution to this problem is to follow the approach taken by POSIX signal handlers and specify that preemptible functions may not call third-party code, but doing so would severely limit their usefulness (Section 2). We opt instead to automatically and dynamically create copies of L to isolate state from different timed functions. Making this approach work on top of existing systems software required solving many design and implementation challenges, which we cover when we introduce *libgotcha* in Section 5.

3.2 Safe concurrency

Automatically handling shared state arising from nonreentrant library interfaces is needed because the sharing is transparent to the programmer. A different problem arises when a programmer explicitly shares state between a preemptible function and any other part of the program. Unlike third-party library authors, this programmer knows they are using preemptible functions, a concurrency mechanism.

When using the C interface, the programmer bears complete responsibility for writing race-free code (e.g., by using atomics and mutexes wherever necessary). The *libinger* Rust API, however, leverages the language's first-class concurrency support to prevent such mistakes from compiling: `launch()`'s signature requires the wrapped function to be `Send` safe (only reference state in a thread-safe manner) [29].

While the Rust compiler rejects all code that shares state unsafely, it is still possible to introduce correctness bugs such as deadlock [30]. For example, a program might block on a mutex held by the preemptible function's caller (recall that invocation is synchronous, so blocking in a preemptible function does not cause it to yield!). It is sometimes necessary to acquire such a mutex, so *libinger* provides a way to do it: The API has an additional function, `pause()`, that is a rough analog of yield. After performing a try-lock operation, a preemptible function can call `pause()` to immediately return to its caller as if it had timed out. The caller can tell whether a function paused via a flag on its continuation.

3.3 Execution stacks

When a preemptible function times out, *libinger* returns a continuation object. The caller might pass this object around the program, which could later `resume()` from a different stack frame. To handle this case, the `launch()` function switches to a new, dedicated stack just before invoking the user-provided function. This stack is then stored in the continuation alongside the register context.

Because of the infeasibility of moving these stacks after a function has started executing, *libinger* currently heap-allocates large 2-MB stacks so it can treat them as having fixed size. To avoid an order of magnitude slowdown from having such large dynamic allocations on the critical path, *libinger* preallocates a pool of reusable stacks when it is first used.

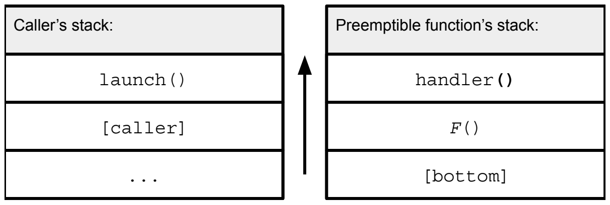


Figure 3: The stacks just before a timeout. Upon discovering that the preemptible function has exceeded its time bound, the handler jumps into the `launch()` (or `resume()`) function, which in turn returns to the original call site, removing its own stack frame in the process.

3.4 Timer interrupts

Whenever *libinger* is executing a user-provided function, we enable fine-grained timer interrupts to monitor that function's elapsed running time. A timer interrupt fires periodically,[3] causing our signal handler to be invoked. If the function exceeds its timeout, this handler saves a continuation by dumping the machine's registers. It then performs an unstructured jump out of the signal handler and back into the `launch()` or `resume()` function, switching back to the caller's stack as it does so. Figure 3 shows the two stacks of execution that are present while the signal handler is running.

A subsequent `resume()` call restores the registers from the stored continuation, thereby jumping back into the signal handler. The handler returns, resuming the preemptible function from the instruction that was executing when the preemption signal arrived.

To support blocking system calls, we use the `SA_RESTART` flag when installing the signal handler to instruct libc to restart system calls that are interrupted by the signal [26]. We direct signals at the process's specific threads that are running preemptible functions by allocating signal numbers from a pool, an approach that limits the number of simultaneous invocations to the number of available signals; this restriction could be lifted by instead using the Linux-specific `SIGEV_THREAD_ID` timer notification feature [31].

3.5 Cancellation

Should a caller decide not to finish running a timed-out preemptible function, it must deallocate it. In Rust, deal-

[3]Signal arrival is accurate to microsecond timescales, but exhibits a warmup effect. For simplicity, we use a fixed signal frequency for all preemptible functions, but this is not fundamental to the design. In the future, we plan to adjust each function's frequency based on its timeout, and to delay the first signal until shortly before the prescribed timeout (in the case of longer-running functions).

location happens implicitly via the `linger_t` type's destructor, whereas users of the C interface are responsible for explicitly calling the *libinger* `cancel()` function.

Cancellation cleans up *libinger* resources allocated by `launch()`; however, the current implementation does not automatically release resources already claimed by the preemptible function itself. While the lack of a standard resource deallocation API makes such cleanup inherently hard to do in C, it is possible in Rust and other languages in which destructor calls are ostensibly guaranteed. For instance, the approach proposed by Boucher et al. [6] could be employed to raise a panic (exception) on the preemptible function's stack. This in turn would cause the language runtime to unwind each stack frame, invoking local variables' destructors in the process.

4 Thread library: *libturquoise*

Until now, we have limited our discussion to synchronous, single-threaded programs. In this section, we will show that the preemptible function abstraction is equally relevant to asynchronous and parallel programs, and that it composes naturally with both futures and threads. As a proof of concept, we have created *libturquoise*,[4] a preemptive userland thread library.

That *libturquoise* provides preemptive scheduling is a significant achievement: *Shinjuku* observes that "there have been several efforts to implement efficient, user-space thread libraries. They all focus on cooperative scheduling" [17]. (Though *RT* from Section 2 could be a counterexample, its lack of nonreentrancy support renders it far from general purpose.) We attribute the dearth of preemptive userland thread libraries to a lack of natural abstractions to support them.

Before presenting the *libturquoise* design, we begin with some context about futures.

4.1 Futures and asynchronous I/O

As mentioned in Section 1, futures are a primitive for expressing asynchronous program tasks in a format amenable to cooperative scheduling. Structuring a program around futures makes it easy to achieve low latency by enabling the runtime to reschedule slow operations off the critical path. Alas, blocking system calls (which cannot be rescheduled by userland) defeat this approach.

The community has done extensive prior work to support asynchronous I/O via result callbacks [19, 18, 20, 23]. Futures runtimes such as Rust's Hyper [16] have adapted this approach by providing I/O libraries whose functions return futures. Rather than duplicate this work, we have integrated preemptible functions with futures so they can leverage it.

[4]so called because it implements "green threading with a twist"

```
function PreemptibleFuture(Future fut,
                           Num timeout):
    function adapt():
        // Poll wrapped future in the usual way
        while poll(fut) == NotReady:
            pause()
    fut.linger = launch(adapt, CREATE_ONLY)
    fut.timeout = timeout
    return fut

// Custom polling logic for preemptible futures
function poll(PreemptibleFuture fut):
    resume(fut.linger, fut.timeout);
    if has_finished(fut.linger):
        return Ready
    else
        if called_pause(fut.linger):
            notify_unblocked(fut.subscribers)
        return NotReady
```

Listing 3: Futures adapter type (pseudocode)

4.2 Preemptible futures

For seamless interoperation between preemptible functions and the futures ecosystem, we built a preemptible future adapter that wraps the *libinger* API. Like a normal future, a preemptible future yields when its result is not ready, but it can also time out.

Each language has its own futures interface, so preemptible futures are not language agnostic like the preemptible functions API. Fortunately, they are easy to implement by using `pause()` to propagate cooperative yields across the preemptive function boundary. We give the type construction and polling algorithm in Listing 3; our Rust implementation is only 70 lines.

4.3 Preemptive userland threading

We built the *libturquoise* thread library by modifying the *tokio-threadpool* [13] work-stealing scheduler from the Rust futures ecosystem. Starting from version 0.1.16 of the upstream project, we added 50 lines of code that wrap each incoming task in a preemptible future.

Currently, *libturquoise* assigns each future it launches or resumes the same fixed time budget, although this design could be extended to support multiple job priorities. When a task times out, the scheduler pops it from its worker thread's job queue and pushes it to the incoming queue, offering it to any available worker for rescheduling after all other waiting jobs have had a turn.

5 Shared state: *libgotcha*

We now present one more artifact, *libgotcha*. Despite the name, it is more like a runtime that isolates hidden shared state within an application. Although the rest of the program does not interact directly with *libgotcha*, its

```
static bool two;
bool three;

linger_t caller(const char *s, u64 timeout) {
    stdout = NULL;
    two = true;
    three = true;
    return launch(timed, timeout, s);
}

void timed(void *s) {
    assert(stdout); // (1)
    assert(two); // (2)
    assert(three); // (3)
}
```

Listing 4: Demo of isolated (1) **vs. shared** (2&3) **state**

presence has a global effect: once loaded into the process image, it employs a technique we call **selective relinking** to dynamically intercept and reroute many of the program's function calls and global variable accesses.

The goal of *libgotcha* is to establish around every preemptible function a memory isolation boundary encompassing whatever third-party libraries that function interacts with (Section 3.1). The result is that the only state shared across the boundary is that explicitly passed via arguments, return value, or closure—the same state the application programmer is responsible for protecting from concurrency violations (Section 3.2). Listing 4 shows the impact on an example program, and Figure 1 classifies libraries by how *libgotcha* supports them.

Note that *libgotcha* operates at runtime; this constrains its visibility into the program, and therefore the granularity of its operation, to shared libraries. It therefore assumes that the programmer will dynamically link all third-party libraries, since otherwise there is no way to tell them apart from the rest of the program at runtime. We feel this restriction is reasonable because a programmer wishing to use *libinger* or *libgotcha* must already have control over their project's build in order to add the dependency.

Before introducing the *libgotcha* API and explaining selective relinking, we now briefly motivate the need for *libgotcha* by demonstrating how existing system interfaces fail to provide the required type of isolation.

5.1 Library copying: namespaces

Expanding a preemptible function's isolation boundary to include libraries requires providing it with private copies of those libraries. POSIX has long provided a `dlopen()` interface to the dynamic linker for loading shared objects at runtime; however, opening an already-loaded library just increments a reference count, and this function is therefore of no use for making copies.

```
typedef long libset_t;

bool libset_thread_set_next(libset_t);
libset_t libset_thread_get_next(void);
bool libset_reinit(libset_t);
```

Listing 5: *libgotcha* C interface

Fortunately, the GNU dynamic linker (`ld-linux.so`) also supports Solaris-style **namespaces**, or isolated sets of loaded libraries. For each namespace, `ld-linux.so` maintains a separate set of loaded libraries whose dependency graph and reference counts are tracked independently from the rest of the program [9].

It may seem like namespaces provide the isolation we need: whenever we `launch(`*F*`)`, we can initialize a namespace with a copy of the whole application and transfer control into that namespace's copy of *F*, rather than the original. The problem with this approach is that it breaks the lexical scoping of static variables. For example, Listing 4 would fail assertion (2).

5.2 Library copying: libsets

We just saw that namespaces provide too much isolation for our needs: because of their completely independent dependency graphs, they never encounter any state from another namespace, even according to normal scoping rules. However, we can use namespaces to build the abstraction we need, which we term a **libset**. A libset is like a namespace, except that the program can decide whether symbols referenced within a libset resolve to the same libset or a different one. Control libraries such as *libinger* configure such **libset switches** via *libgotcha*'s private control API, shown in Listing 5.

This abstraction serves our needs: when a `launch(`*F*`)` happens, *libinger* assigns an available `libset_t` exclusively to that preemptible function. Just before calling *F*, it informs *libgotcha* by calling `libset_thread_set_next()` to set the thread's **next libset**: any dynamic symbols used by the preemptible function will resolve to this libset. The thread's **current libset** remains unchanged, however, so the preemptible function itself executes from the same libset as its caller and the two share access to the same global variables.

One scoping issue remains, though. Because dynamic symbols can resolve back to a definition in the same executable or shared object that used them, Listing 5 would fail assertion (3) under the described rules. We want global variables defined in *F*'s object file to have the same scoping semantics regardless of whether they are declared `static`, so *libgotcha* only performs a namespace switch when the use of a dynamic symbol occurs in a different executable or shared library than that symbol's definition.

5.3 Managing libsets

At program start, *libgotcha* initializes a pool of libsets, each with a full complement of the program's loaded object files. Throughout the program's run *libinger* tracks the libset assigned to each preemptible function that has started running but not yet reached successful completion. When a preemptible function completes, *libinger* assumes it has not corrupted its libset and returns it to the pool of available ones. However, if a preemptible function is canceled rather than being allowed to return, *libinger* must assume that its libset's shared state could be corrupted. It unloads and reloads all objects in such a libset by calling `libset_reinit()`.

While *libinger* in principle runs on top of an unmodified `ld-linux.so`, in practice initializing more than one namespace tends to exhaust the statically-allocated thread-local storage area. As a workaround, we build glibc with an increased `TLS_STATIC_SURPLUS`. It is useful to also raise the maximum number of namespaces by increasing `DL_NNS`.

5.4 Selective relinking

Most of the complexity of *libgotcha* lies in the implementation of selective relinking, the mechanism underlying libset switches.

Whenever a program uses a dynamic symbol, it looks up its address in a data structure called the global offset table (GOT). As it loads the program, `ld-linux.so` eagerly resolves the addresses of all global variables and some functions and stores them in the GOT.

Selective relinking works by shadowing the GOT.[5] As soon as `ld-linux.so` finishes populating the GOT, *libgotcha* replaces every entry that should trigger a libset switch with a fake address, storing the original one in its shadow GOT, which is organized by the libset that houses the definition. The fake address used depends upon the type of symbol:

Functions' addresses are replaced by the address of a special function, `procedure_linkage_override()`. Whenever the program tries to call one of the affected functions, this intermediary checks the thread's next libset, looks up the address of the appropriate definition in the shadow GOT, and jumps to it. Because `procedure_linkage_override()` runs between the caller's `call` instruction and the real function, it is written in assembly to avoid clobbering registers. Instead of being linked to their symbol definitions at load time, some function calls resolve lazily the first time they are called: their GOT entries initially point to a special lookup function in the dynamic linker that rewrites the GOT entry when invoked. Such memoization would remove our intermediary, so we alter the ELF relocation entries of affected symbols to trick the dynamic linker into updating our shadow GOT instead.

[5]Hence the name *lib**got**cha*.

Global variables' addresses are replaced with a unique address within a mapped but inaccessible page. When the program tries to read or write such an address, a segmentation fault occurs; *libgotcha* handles the fault, disassembles the faulting instruction to determine the base address register of its address calculation,[6] loads the address from this register, computes the location of the shadow GOT entry based on the fake address, checks the thread's next libset, and replaces the register's contents with the appropriate resolved address. It then returns, causing the faulting instruction to be reexecuted with the valid address this time.[7]

5.5 Uninterruptible code: uncopyable

The library-copying approach to memory isolation works for the common case, and allows us to handle most third-party libraries with no configuration. However, in rare cases it is not appropriate. The main example is the `malloc()` family of functions: in Section 1, we observed that not sharing a common heap complicates ownership transfer of objects allocated from inside a preemptible function. To support dynamic memory allocation and a few other special cases, *libgotcha* has an internal whitelist of **uncopyable** symbols.

From *libgotcha*'s perspective, uncopyable symbols differ only in what happens on a libset switch. If code executing in any libset other than the application's **starting libset** calls an uncopyable symbol, a libset switch still occurs, but it returns to the starting libset instead of the next libset; thus, all calls to an uncopyable symbol are routed to a single, globally-shared definition. When the function call that caused one of these special libset switches returns, the next libset is restored to its prior value. The *libgotcha* control API provides one more function, `libset_register_interruptible_callback()`, that allows others to request a notification when one of these libset restorations occurs.

Because it is never safe to preempt while executing in the starting libset, the first thing the *libinger* preemption handler described in Section 3.4 does is check whether the thread's next libset is set to the starting one; if so, it disables preemption interrupts and immediately returns. However, *libinger* registers an interruptible call-

[6]Although it is possible to generate code sequences that are incompatible with this approach (e.g., because they perform in-place pointer arithmetic on a register rather than using displacement-mode addressing with a base address), we employ a few heuristics based on the context of the instruction and fault; in our experience, these cover the common cases.

[7]This does not break applications with existing segfault handlers: we intercept their calls to `sigaction()`, and forward the signal along to their handler when we are unable to resolve an address ourselves.

back that it uses to reenable preemption as soon as any uncopyable function returns.

5.6 Limitations

The current version of *libgotcha* includes partial support for thread-local storage (TLS). Like other globals, TLS variables are copied along with the libset; this behavior is correct because a thread might call into the same library from multiple preemptible functions. However, we do not yet support migrating TLS variables between threads along with their preemptible function. This restriction is not fundamental: the TLS models we support (general dynamic and local dynamic) use a support function called `__tls_get_addr()` to resolve addresses [10], and *libgotcha* could substitute its own implementation that remapped the running thread's TLS accesses to that of the preemptible function's initial thread when executing outside the starting libset.

While selective relinking supports the ordinary `GLOB_DAT` (eager) and `JUMP_SLOT` (lazy) ELF dynamic relocation types, it is incompatible with the optimized `COPY` class of dynamic variable relocations. The `COPY` model works by allocating space for all libraries' globals in the executable, enabling static linking from the program's code (but not its dynamic libraries'). This transformation defeats selective relinking for two reasons: the use of static linking prevents identifying symbol uses in the executable, and the cross-module migration causes breakages such as failing assertion (3) from Listing 4. When building a program that depends on *libgotcha*, programmers must instruct their compiler to disable `COPY` relocations, as with the `-fpic` switch to GCC and Clang. If *libgotcha* encounters any `COPY` relocations in the executable, it prints a load-time warning.

Forsaking `COPY` relocations does incur a small performance penalty, but exported global variables are rare now that thread safety is a pervasive concern in system design. Even the POSIX-specified `errno` global is gone: the Linux Standard Base specifies that its address is resolved via a call to the `__errno_location()` helper function [21].

5.7 Case study: auto async-signal safety

We have now described the role of *libgotcha*, and how *libinger* uses it to handle nonreentrancy. Before concluding our discussion, however, we note that *libgotcha* has other interesting uses in its own right.

As an example, we have used it to implement a small library, *libas-safe*, that transparently allows an application's signal handlers to call functions that are not async-signal safe, which is forbidden by POSIX because it is normally unsafe.

Written in 127 lines of C, *libas-safe* works by injecting code before `main()` to switch the program away from its starting libset. It shadows the system's `sigaction()`, providing an implementation that:

- Provides copy-based library isolation for signal handlers by switching the thread's next libset to the starting libset while a signal handler is running.
- Allows use of uncopyable code such as `malloc()` from a signal handler by deferring signal arrival whenever the thread is already executing in the starting libset, then delivering the deferred signal when the interruptible callback fires.

In addition to making signal handlers a lot easier to write, *libas-safe* can be used to automatically "fix" deadlocks and other misbehaviors in misbehaved signal-handling programs just by loading it via `LD_PRELOAD`.

We can imagine extending *libgotcha* to support other use cases, such as simultaneously using different versions or build configurations of the same library from a single application.

Operation	Duration (μs)
`launch()`	4.6 ± 0.05
`resume()`	4.4 ± 0.02
`cancel()`	4767.7 ± 1168.7
`fork()`	207.5 ± 79.3
`pthread_create()`	32.5 ± 8.0

Table 2: Latency of preemptible function interface

6 Evaluation

We now evaluate preemptible function performance, presenting several microbenchmarks and two examples of their application to improve existing systems' resilience to malicious or otherwise long-running requests. All experiments were run on an Intel Xeon E5-2683 v4 (Broadwell) server running Linux 4.12.6, rustc 1.36.0, gcc 9.2.1, and glibc 2.29.

6.1 Microbenchmarks

Table 2 shows the overhead of *libinger*'s core functions. Each test uses hundreds of preemptible functions, each with its own stack and continuation, but sharing an implementation; the goal is to measure invocation time, so the function body immediately calls `pause()`. For comparison, we also measured the cost of calling `fork()` then `exit()`, and of calling `pthread_create()` with an empty function, while the parent thread waits using `waitpid()` or `pthread_join()`, respectively.

The results show that, as long as preemptible functions are eventually allowed to run to completion, they are an order of magnitude faster than spawning a thread and two orders of magnitude faster than forking a process. Although cancellation takes milliseconds in the benchmark application, this operation need not lie on the critical path unless the application is cancelling tasks frequently enough to exhaust its supply of libsets.

Recall that linking an application against *libgotcha* imposes additional overhead on most dynamic symbol accesses; we report these overheads in Table 3a. Eager function calls account for almost all of a modern program's dynamic symbol accesses: lazy resolution only occurs the first time a module calls a particular function (Section 5.4) and globals are becoming rare (Section 5.6).

Table 3b shows that the *libgotcha* eager function call overhead of 14 ns is on par with the cost of a trivial C library function (`gettimeofday()`) and one-third that of a simple system call (`getpid()`). This overhead affects the entire program, regardless of the current libset at the time of the call. Additionally, calls to uncopyable functions from within a preemptible function incur several extra nanoseconds of latency to switch back to the main namespace as described in Section 5; Table 3c breaks this overhead down to show the cost of notification callbacks at the conclusion of such a call (always required by *libinger*).

6.2 Web server

To test whether our thread library could combat head-of-line blocking in a large system, we benchmarked *hyper*, the highest-performing Web server in TechEmpower's plaintext benchmark as of July 2019 [16]. The server uses *tokio-threadpool* for scheduling; because the changes described in Section 4 are transparent, making *hyper* preemptive was as easy as building against *libturquoise* instead. In fact, we did not even check out the *hyper* codebase. We configured *libturquoise* with a task timeout of 2 ms, give or take a 100-μs *libinger* preemption interval, and configured it to serve responses only after spinning in a busy loop for a number of iterations specified in each request. For our client, we modified version 4.1.0 of the *wrk* [14] closed-loop HTTP load generator to separately record the latency distributions of two different request classes.

Our testbed consisted of two machines connected by a direct 10-GbE link. We pinned *hyper* to the 16 physical cores on the NIC's NUMA node of our Broadwell server. Our client machine, a Intel Xeon E5-2697 v3 (Haswell) running Linux 4.10.0, ran a separate *wrk* process pinned to each of the 14 logical cores on the NIC's NUMA node. Each client core maintained two concurrent pipelined HTTP connections.

We used loop lengths of approximately 500 μs and 50 ms for short and long requests, respectively, viewing the latter requests as possible DoS attacks on the system. We varied the percentage of long requests from 0% to 2% and measured the round-trip median and tail latencies of short requests and the throughput of all requests. Figure 4 plots the results for three server configurations: `baseline` is cooperative scheduling via *tokio-threadpool*, `baseline+libgotcha` is the same but with *libgotcha* loaded to assess the impact of slower dynamic function calls, and `baseline+libturquoise` is preemptive scheduling via *libturquoise*. A 2% long request mix was enough to reduce the throughput of the *libgotcha* server enough to impact the median short request latency. The experiment shows that preemptible functions keep the tail latency of short requests scaling linearly at the cost of a modest 4.5% median latency overhead when not under attack.

6.3 Image decompression

The Web benchmark showed preemptive scheduling at scale, but did not exercise preemptible function cancellation. To demonstrate this feature, we consider decompression bombs, files that expand exponentially when decoded, consuming enormous computation time in addition to their large memory footprint. PNG files are vulnerable to such an attack, and although *libpng* now supports some mitigations [24], one cannot always expect (or trust) such functionality from third-party code.

We benchmarked the use of *libpng*'s "simple API" to decode an in-memory PNG file. We then compared against synchronous isolation using preemptible functions, as well as the naïve alternative mitigations proposed in Section 1. For preemptible functions, we wrapped all uses of *libpng* in a call to `launch()` and used a dedicated (but blocking) reaper thread to remove the cost of cancellation from the critical path; for threads, we used `pthread_create()` followed by `pthread_timedjoin_np()` and, conditionally, `pthread_cancel()` and `pthread_join()`; and for processes, we used `fork()` followed by `sigtimedwait()`, a conditional `kill()`, then a `waitpid()` to reap the child. We ran `pthread_cancel()` both with and without asynchronous cancelability enabled, but the former always deadlocked. The timeout was 10 ms in all cases.

Running on the benign RGB image `mirjam_meijer_mirjam_mei.png` from version `1:0.18+dfsg-15` of Debian's `openclipart-png` package showed `launch()` to be both faster and lower-variance than the other approaches, adding 355 μs or 5.2% over the baseline (Figure 5a). The results for `fork()` represent a best-case scenario for that technique, as we did not implement a shared memory mechanism for sharing the buffer, and the cost of the system call will increase with the number pages mapped by the process (which was small in this case).

Next, we tried a similarly-sized RGB decompression bomb from revision `b726584` of `https://bomb.codes` (Figure 5b). Without asynchronous cancelability, the pthreads approach was unable to interrupt the thread. Here, `launch()` exceeded the deadline by just 100 μs, a

Symbol resolution scheme	Time without *libgotcha* (*ns*)	Time with *libgotcha* (*ns*)
eager (load time)	2 ± 0	14 ± 0
lazy (runtime)	100 ± 1	125 ± 0
global variable	0 ± 0	3438 ± 13

(a) Generic symbols, without and with *libgotcha*

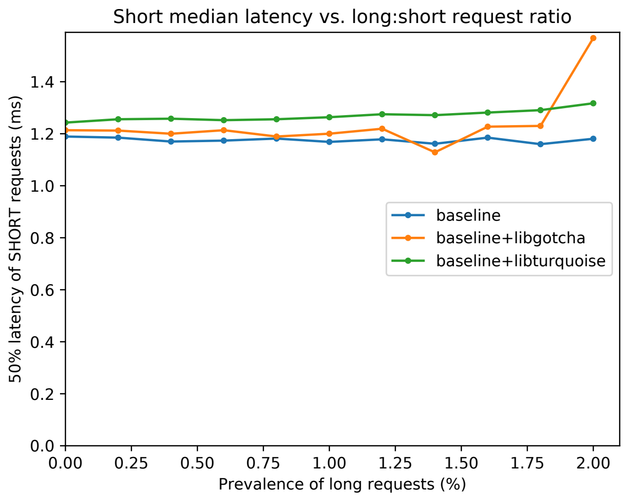
Baseline	Time without *libgotcha* (*ns*)
`gettimeofday()`	19 ± 0
`getpid()`	44 ± 0

(b) Library functions and syscalls without *libgotcha*

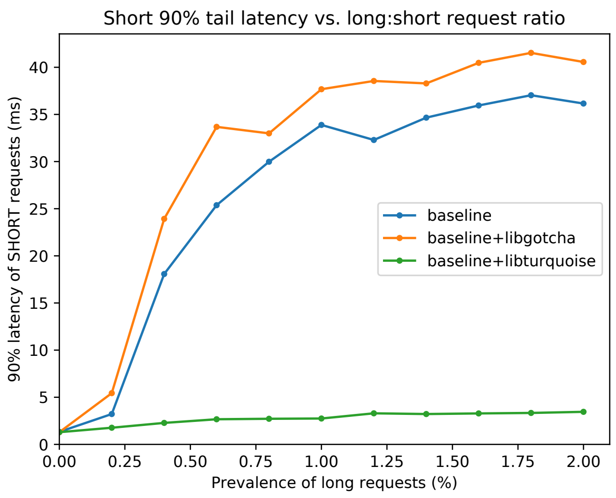
Trigger	Time with *libgotcha* (*ns*)
Uncopyable call	21 ± 0
Uncopyable call + callback	25 ± 0

(c) Uncopyable calls triggering a libset switch

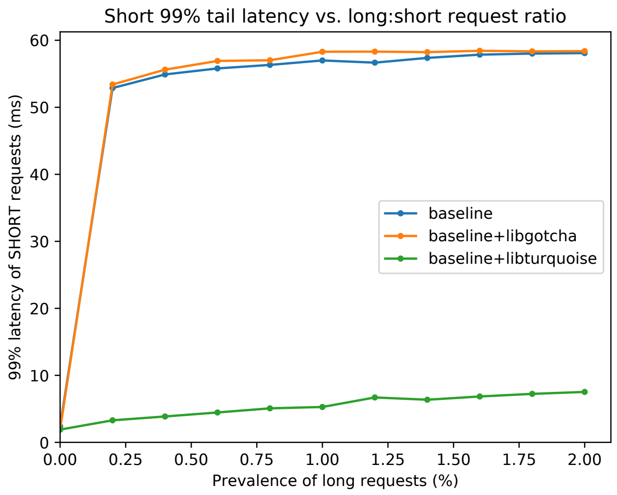
Table 3: Runtime overheads of accessing dynamic symbols



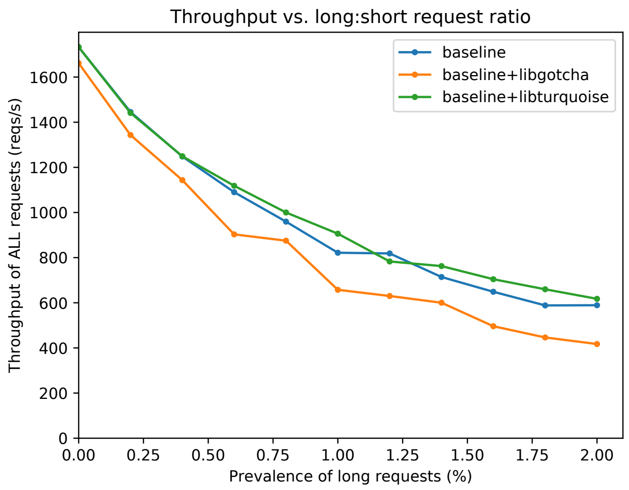
(a) Median latency



(b) 90% tail latency

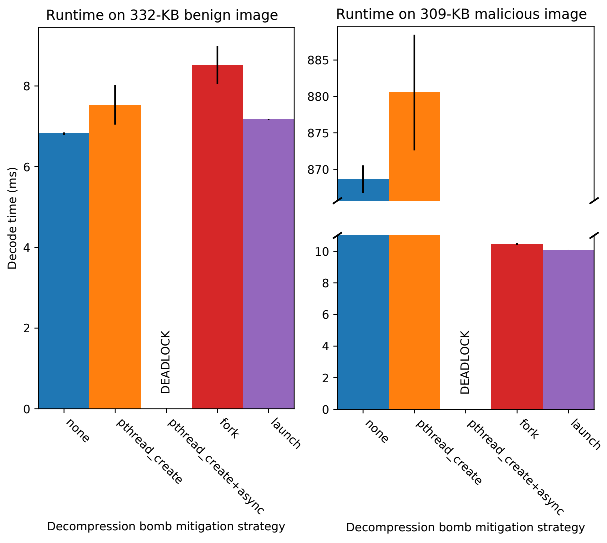


(c) 99% tail latency



(d) Overall throughput

Figure 4: *hyper* Web server with 500-μs (short) and 50-ms (long) requests



(a) Benign image **(b) Malicious image**

Figure 5: *libpng* in-memory image decode times

figure that includes deviation due to the 100-μs preemption interval in addition to *libinger*'s own overhead. It again had the lowest variance.

Applying preemptible functions proved easy: the `launch()/cancel()` approach took just 20 lines of Rust, including the implementation of a reaper thread to move libset reinitialization off the critical path. In comparison, the `fork()/sigtimedwait()` approach required 25 lines of Rust. Note that both benchmarks include unsafe Rust (e.g., to use the *libpng* C library and zero-copy buffers).

7 Future work

One of our contributions is asynchronous cancellation, something rarely supported by the state of the art. In Section 3.5, we noted our lack of support for automated resource cleanup; however, we outlined a possible approach for languages such as Rust, which we intend to investigate further. Cancellation is currently our most expensive operation because of the libset reinitialization described in Section 5.3, but we plan to improve this by restoring only the writeable regions of each module.

Another area for improvement is signal-handling performance optimization: whereas *Shinjuku* is able to preempt every 5 μs with a 10% throughput penalty [17], we have observed a similar throughput drop while only preempting every 20 μs via our technique [6]. We have not yet heavily optimized *libinger*, and have reason to believe that doing so will allow our design to achieve a preemption granularity midway between those figures for the same throughput cost. Because *Shinjuku* executes in privilege ring 0, they preempt by issuing interprocessor interrupts (IPIs) directly rather than using Linux signals. Their microbenchmarks reveal an IPI:signal latency ratio of roughly 1:2.5 (1,993 vs. 4,950 CPU cycles), indicating that we are not achieving peak performance. Furthermore, a key design difference between our systems suggests that this ratio probably understates the performance we could achieve. In their benchmark, roughly 42% of cycles are spent sending each signal, a cost we can amortize because our design uses recurring timer signals to counter warmup effects. A further 6.9% of benchmarked cycles are spent propagating the signal between cores, which should not affect our system because we request the timer signals on the same core that will receive them rather than using a central watchdog thread to preempt all workers. Context switching is likely responsible for most of our unexpected latency: by writing our signal handler very carefully, we should be able to adopt the same optimizations they describe (skipping signal mask and floating-point register swaps).

The selective relinking technique that underlies our interface allows safe pausing and cancellation in the presence of shared state, *independent of preemption mechanism*. In lieu of a timeout, control transfer might result from another scheduling consideration, such as real-time computing or task priority.

8 Conclusion

We presented the lightweight preemptible function, a new composable abstraction for invoking a function with a timeout. This enabled us to build a first-in-class preemptive userland thread library by implementing preemption atop a cooperative scheduler, rather than the other way around. Our evaluation shows that lightweight preemptible functions have overheads of a few percent (lower than similar OS primitives), yet enable new functionality.

We believe the lightweight preemptible function abstraction naturally supports common features of large-scale systems. For example: In support of graceful degradation, a system might use a preemptible function to abort the rendering of a video frame in order to ensure SLA adherance. An RPC server might preserve work by processing each request in a preemptible function and memoizing the continuations; if a request timed out but was later retried by the client, the server could resume executing from where it left off.

Acknowledgements

This work was supported by the U.S. National Science Foundation under award CNS–1700521, as well as the Intel Science and Technology Center for Visual Cloud Systems. The authors thank Holden and the other members of the Pittsburgh Rust Coffee meetup for early implementation discussions about the safe use of unstructured control flow in Rust programs. We also thank our shepherd, Tim Harris, for his helpful suggestions.

References

[1] The Go programming language. https://golang.org, 2019.

[2] The Rust programming language. https://www.rust-lang.org, 2019.

[3] 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. In *Proceedings of the thirteenth ACM symposium on operating system principles (SOSP '91)*, 1991.

[4] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe. A fork() in the road. In *HotOS '19: Proceedings of the workshop on hot topics in operating systems*, May 2019.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazères, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementaiton (OSDI'12)*, 2012.

[6] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference*, Boston, MA, 2018.

[7] A. Clements. Go runtime: tight loops should be preemptible. https://github.com/golang/go/issues/10958, 2015.

[8] T. Delisle. Concurrency in C∀, 2018. URL https://uwspace.uwaterloo.ca/handle/10012/12888.

[9] dlmopen. dlmopen(3) manual page from Linux man-pages project, 2019.

[10] U. Drepper. ELF handling for thread-local storage. Technical report, 2013. URL https://akkadia.org/drepper/tls.pdf.

[11] D. Eloff. Go proposal: a faster C-call mechanism for non-blocking C functions. https://github.com/golang/go/issues/16051, 2016.

[12] G. H. et al. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research Technical Reports, 2005. URL https://www.microsoft.com/en-us/research/publication/an-overview-of-the-singularity-project.

[13] GitHub. Tokio thread pool. https://github.com/tokio-rs/tokio/tree/tokio-threadpool-0.1.16/tokio-threadpool, 2019.

[14] GitHub. wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk, 2019.

[15] C. T. Haynes and D. P. Friedman. Engines build process abstractions. Technical Report TR159, Indiana University Computer Science Technical Reports, 1984. URL https://cs.indiana.edu/ftp/techreports/TR159.pdf.

[16] hyper. hyper: Fast and safe HTTP for the Rust language. https://hyper.rs, 2019.

[17] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for microsecond-scale tail latency. In *Proc. 16th USENIX NSDI*, Boston, MA, Feb. 2019.

[18] libev. libev. http://libev.schmorp.de.

[19] libevent. libevent. https://libevent.org.

[20] libuv. libuv: Cross-platform asynchronous I/O. https://libuv.org.

[21] Linux Standard Base Core specification. __errno_location. http://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/baselib---errno-location.html, 2015.

[22] M. S. Mollison and J. H. Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, PA, 2013.

[23] mordor. mordor: A high-performance I/O library based on fibers. https://github.com/mozy/mordor.

[24] PNG reference library: libpng. Defending libpng applications against decompression bombs. https://libpng.sourceforge.io/decompression_bombs.html, 2010.

[25] pthread_setcanceltype(3). pthread_setcanceltype() manual page from Linux man-pages project, 2017.

[26] sigaction. sigaction(2) manual page from the Linux man-pages project, 2019.

[27] signal-safety. signal-safety(7) manual page from Linux man-pages project, 2019.

[28] TerminateThread. TerminateThread function. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminatethread, 2018.

[29] The Rust Programming Language. Extensible Concurrency with the Sync and Send Traits. https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html, 2019.

[30] The Rust Reference. Behavior not considered unsafe. https://doc.rust-lang.org/stable/reference/behavior-not-considered-unsafe.html, 2019.

[31] timer_create. timer_create(2) manual page from the Linux man-pages project, May 2020.

[32] C. J. Vanderwaart. Static enforcement of timing policies using code certification. Technical Report CMU-CS-06-143, Carnegie Mellon Computer Science Technical Report Collection, 2006. URL http://reports-archive.adm.cs.cmu.edu/anon/2006/abstracts/06-143.html.