Lightweight Preemptible Functions
A thesis proposal

Sol Boucher
April 16, 2020

Thesis committee:
David G. Andersen, chair
Adam Belay
Michael Kaminsky
Brandon Lucia

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
# Table of Contents

1 **Introduction** .................................................. 1  
   1.1 Thesis statement .................................................. 2  
   1.2 Structure .......................................................... 2  

2 **Function calls with timeouts** .................................. 4  
   2.1 Related work ....................................................... 5  
   2.2 Timed functions: *libinger* ..................................... 7  
      2.2.1 Automatic handling of shared state ....................... 8  
      2.2.2 Safe concurrency ............................................. 9  
      2.2.3 Execution stacks ............................................ 9  
      2.2.4 Timer interrupts ........................................... 10  
      2.2.5 Cancellation ............................................... 10  
   2.3 Evaluation ....................................................... 11  
      2.3.1 Microbenchmarks ............................................. 11  
      2.3.2 Image decompression ....................................... 11  

3 **Nonreentrancy and selective relinking** ....................... 14  
   3.0.1 Library copying: namespaces ............................... 14  
   3.0.2 Library copying: libsets ................................... 15  
   3.0.3 Managing libsets ........................................... 16  
   3.0.4 Selective relinking ......................................... 16  
   3.0.5 Uninterruptible code: uncopyable ......................... 17  
   3.0.6 Case study: auto async-signal safety ..................... 17  
   3.1 Evaluation ....................................................... 18  

4 **Preemptive userland threading** ................................ 20  
   4.0.1 Preemptible futures ......................................... 20  
   4.1 Evaluation ....................................................... 21  

5 **Microsecond-scale microservices** ............................. 23  
   5.1 Introduction ..................................................... 23  
   5.2 Motivation ....................................................... 24  
      5.2.1 Process-level isolation is too slow ..................... 25  
      5.2.2 Intra-process preemption is fast ........................ 26  
   5.3 Providing Isolation ............................................ 27
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>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 (e.g., F) may be defined directly in your program, or in some other loaded library.</td>
</tr>
<tr>
<td>2.2</td>
<td>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.</td>
</tr>
<tr>
<td>2.3</td>
<td>libpng in-memory decode times for two different images</td>
</tr>
<tr>
<td>4.1</td>
<td>hyper Web server with 500-µs (short) and 50-ms (long) requests</td>
</tr>
<tr>
<td>5.1</td>
<td>Language-based isolation design. The dispatcher process uses shared in-memory queues to feed requests to the worker processes, each of which runs one user-supplied microservice at a time.</td>
</tr>
<tr>
<td>5.2</td>
<td>Effect of SIGALRM quantum on hashing tput.</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Number</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Systems providing timed code at sub-process granularity</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Latency of preemptible function interface</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Runtime overheads of accessing dynamic symbols</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Microservice invocation performance</td>
<td>26</td>
</tr>
</tbody>
</table>
List of Listings

2.1 Preemptible functions core interface .................................................. 7
2.2 Preemptible function usage example ..................................................... 7
3.1 Demo of isolated (1) vs. shared (2&3) state ........................................... 15
3.2 libgotcha C interface ........................................................................... 15
4.1 Futures adapter type (pseudocode) ...................................................... 21
Chapter 1

Introduction

The abstraction most fundamental to modern programs is the function, a section of code that expects zero or more data inputs, performs some computation, and produces zero or more outputs. It is a structured control flow primitive that obeys a strict convention: whenever invoked from one of its call sites, a function runs from beginning to (one possible) end, at which point execution resumes in the caller just after the call site. It is also a synchronous primitive; that is, all these steps happen sequentially and in order. Because processors conceptually implement synchronous computation, scheduling a function is as trivial as instructing the processor to jump from the call site to its starting address, then jump back to the (saved) address subsequent to the call site. Thus, the program continues executing throughout, with no inherent need for intervention by an external scheduler or other utility software.

Note, however, that just because the program has retained control does not mean the programmer has. Precisely because functions represent an abstraction, the programmer who calls one is not necessarily familiar with its specific implementation. This can make it hard to predict the function’s duration, yet calling it requires the programmer to trust it to eventually finish and relinquish control. The programmer may have made a promise (e.g., a service-level agreement) that their whole program will complete within a specified timeframe; unfortunately, they cannot certify their compliance without breaking the abstraction and examining the internals of each function they call. Even then, untrusted or unpredictable input may make the function’s performance characteristics unclear: Perhaps it solves a problem that is known to be intractable for certain cases, but such inputs are difficult to identify a priori. Perhaps it performs format decoding or translation that is susceptible to attacks such as decompression bombs. Or perhaps it simply contains bug that open it to inefficient corner cases or even an infinite loop.

Faced with such problems, the programmer is often tempted to resort to an asynchronous invocation strategy, whereby the function runs in the background while the programmer maintains control of the rest of the program. Common abstractions following this model include the operating system’s own processes and threads, as well as the threads, coroutines, and futures (i.e., promises) provided by some libraries and language runtimes. Any use of asynchronous computation requires an external scheduler to allocate work.

Here again, the programmer is sacrificing control. Upon handing execution control to a scheduler, dependencies are no longer clear from the program’s structure and must be passed to the scheduler by encoding them in synchronization constructs; however, it is difficult to fully communicate the relevant bits of the application logic across this abstraction boundary, which can result
in unintended resource-sharing effects such as priority inversion. Furthermore, each software scheduler is itself a piece of code, and because its job does not represent useful application work, any time it spends executing is pure overhead. Therefore, introducing unnecessary scheduling necessarily reduces per-processor performance.

In many cases, the only tool necessary to ensure timely completion of a program is preemption. Instead of confronting this directly, current programming environments incentivize the programmer to rely on a scheduler to fix the problem, limiting them to whatever coarse timescales (often milliseconds) the OS scheduler operates at, or (in the case of userland schedulers) to cooperative scheduling that doesn’t even address the problem of infinite loops. The goal of this work is to design and prototype an interface that extends the programming model with simple preemption, thereby allowing the use of functions without having to break the abstraction and examine their implementations. If the function times out, it is paused so that the programmer can later resume and/or cancel it at the appropriate time. Note that such an interface is still inherently concurrent; indeed, it is the programmer who expresses the schedule describing when to devote time to the timed code, and how much.

1.1 Thesis statement

Modern operating systems provide task preemption as a resource sharing mechanism: when the total number of threads exceeds the number of processors, the kernel scheduler preempts long-running or low-priority threads to allow others to run. Preemption is also useful to applications in its own right, and its interface influences the structure and architecture of such programs. Providing only an asynchronous interface encourages the programmer to leave even simple scheduling to the operating system, thereby accepting the scheduler’s overhead and coarse resolution. We introduce a novel abstraction for preemption at the granularity of a synchronous function call, and demonstrate that this represents a more efficient and composable interface that enables new functionality for latency-critical applications, while being both compatible with the existing systems stack and expressive enough to encode classic asynchrony primitives.

1.2 Structure

The rest of the chapters of this thesis break down this research as follows:

- We prototype our abstraction on top the GNU/Linux operating system with an unmodified kernel (Chapter 2). Although the current implementation is limited to the x86-64 architecture and relies on POSIX features such as signals, timers, and contexts and GNU dynamic linker namespaces, we believe it could be ported to other systems and architectures with additional engineering effort. We examine the performance properties of preemptible function invocations, pauses, resumes, and cancellations.

- Supporting existing nonreentrant code requires an approach we call selective relinking; our implementation includes a lightweight runtime to do this transparently with modest overhead (Chapter 3). We present the technique, along with its interaction with dynamic linking and loading, in detail and with examples of semantic implications.
• Although lightweight preemptible functions are fundamentally synchronous, we demonstrate their expressiveness by applying them to the asynchrony problem of making an existing cooperative thread library preemptive (Chapter 4). We demonstrate the applicability of this artifact to mitigating head-of-line blocking in a modern Web server.

• We discuss the relevance of preemptible functions to serverless computing, where we argue they could be used to accelerate the launch times of microservices, enabling customers to better leverage the low latency of modern datacenter networks and the steady performance improvements of FaaS providers’ walled garden services (Chapter 5).
Chapter 2

Function calls with timeouts

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 translates this to a yield if the result is not yet available.

Key to the popularity of this concurrency abstraction is the ease and seamlessness of parallelizing it. Underlying every major futures runtime is a 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 [5]), however, this logical threading model renders the operating system unaware of individual tasks, meaning context switches are purely cooperative.

We propose an abstraction that is not available in modern languages: an easy way of preemptively scheduling functions (e.g., calling a function with a timeout). An implementation of such preemptible functions based on OS threads suggests itself, where the function would run on a new thread that could be cancelled upon timeout. Unfortunately, the existence of nonreentrant functions makes cancelling a thread 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” [37]. 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 [40].

One might instead seek to implement preemptible functions via the UNIX fork() function. Even ignoring the high overhead, this would require careful configuration of shared memory to ensure that objects created outside the function were accessible inside, and worse yet, that allocations performed inside remained available after it exited. 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, that process can experience the same problems as with the thread approach [7].
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 and limit composability by increasing the abstraction’s complexity. We observe that, when calling a function with a timeout, it is concurrency alone—and not parallelism—that is fundamental.

In this paper, we propose lightweight preemptible functions, a userland abstraction for making function calls with a timeout that differs significantly from the state of the art: (1) Unlike threading abstractions such as coroutines, it runs functions synchronously on the caller’s thread. (2) It supports preemption at granularities in the tens of microseconds, orders of magnitude finer than contemporary OS schedulers’ millisecond scale. (3) It is language agnostic, being implemented in userland without relying on any particular compiler or managed runtime.

### 2.1 Related work

A number of past projects (Table 2.1) have sought to provide bounded-time execution of chunks of code at sub-process 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 what form this takes 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 [2] or excessively-tight loops [45], 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 [22]. This added 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

---

### Table 2.1: Systems providing timed code at sub-process granularity

<table>
<thead>
<tr>
<th>System</th>
<th>Preemptive</th>
<th>Synchronous</th>
<th>Dependencies</th>
<th>Third-party code support</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scheme engines</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Lilt</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>goroutines</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>RT library</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Shinjuku</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>libinger</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

✓ = 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 function must be written in Go and have no foreign dependencies
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 authors mention timer interrupts as an alternative, this was apparently never tried.

Lilt [45] 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 [17].

Some recent languages offer explicit userland threading, which could be used to support timed code. One example is the Go language’s 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 [11].

The solutions described thus far all assume languages with a heavyweight, garbage-collected runtime. However, some recent systems seek to support timed code with fewer dependencies. One example is the userland C-language thread library for realtime systems (here, “RT”) developed by Mollison and Anderson [34], which performs 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. This lightweight runtime achieves average overall scheduling latencies in the tens of microseconds; however, as explained later in this section, the compromise is developer usability.

Shinjuku [26] is an operating system designed to perform preemption at microsecond scale. Built on the Dune framework [8], 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 5.1, 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. Due to its focus on realtime embedded systems, RT assumes that the timed code in its threads will avoid shared state; this 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. And although Go is able to preempt goroutines written in the language itself, a goroutine that makes any foreign calls to other languages is treated as nonpreemptible by the runtime’s scheduler [16].
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 2.1: 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 2.2: Preemptible function usage example

2.2 Timed functions: libinger

To address the literature’s shortcomings, we have developed libinger\footnote{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.”}, a library providing a small API for timed function dispatch (Listing 2.1):

- `launch()` invokes an ordinary function $F$ with an execution time cap of $T$. The call to `launch()` returns when $F$ completes, or after approximately $T$ microseconds if $F$ has not returned by then. In the latter case, `libinger` returns an opaque continuation object recording the intermediate 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 2.2 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
In accordance with our goal of language agnosticism, *libinger* exposes both C and Rust 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 2.1 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 now describe the implementation of *libinger*, beginning with shared state handling.

### 2.2.1 Automatic handling of shared state

As we found in Section 5.1 a key design challenge facing *libinger* is the shared state problem: Suppose a preemptible function $F_0$ calls a stateful routine in a third-party library $L$, and that $F_0$ 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 constituting a concurrency violation, *libinger* must hide state modifications in $L$ by $F_0$ from the execution of $F'_0$ to avoid undefined behavior.

The problem is actually even worse. Those familiar with POSIX signals may notice that, upon a function’s timeout, its caller interrupts it. The rest of the program can therefore be viewed as
a signal handler, and would normally be expected to restrict itself to calling async-signal-safe (roughly, nonreentrant) functions \[39\].

One non-solution to this problem is to instead prevent preemptible functions from calling into third-party code, but doing so would severely limit their usefulness (Section 2.1). We opt instead to automatically and dynamically create copies of $L$ to isolate state from different timed functions. Making this work on top of existing systems software required solving many design and implementation challenges, which we cover when we introduce libgotcha in Section 3.

2.2.2 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) \[41\].

While the Rust compiler rejects all code that shares state unsafely, it is still possible to introduce correctness bugs such as deadlock \[43\]. One way to do this is to 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 this function to immediately return to its caller as if it had timed out. The caller can detect this by checking a flag on the continuation.

2.2.3 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 call `resume()` from a different stack frame. Thus, the continuation must contain not only the register context, but also the stack frames belonging to the preemptible function and its callees. The `launch()` function enables this by switching to a new, dedicated stack just before invoking the user-provided function.

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.
2.2.4 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 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 stacks as it does so. Figure 2.2 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.

2.2.5 Cancellation

Should a caller decide not to finish running a timed-out preemptible function, it must deallocate it. In Rust this 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 the 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 this inherently hard to do in C, it is possible in languages such as Rust that support destructors. For instance, the approach proposed by Boucher et al. [9] could be employed to raise a panic (exception) on the

---

2 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 time limit (in the case of longer-running functions).
<table>
<thead>
<tr>
<th>Operation</th>
<th>Duration (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>launch()</td>
<td>4.6 ± 0.05</td>
</tr>
<tr>
<td>resume()</td>
<td>4.4 ± 0.02</td>
</tr>
<tr>
<td>cancel()</td>
<td>4767.7 ± 1168.7</td>
</tr>
<tr>
<td>fork()</td>
<td>207.5 ± 79.3</td>
</tr>
<tr>
<td>pthread_create()</td>
<td>32.5 ± 8.0</td>
</tr>
</tbody>
</table>

Table 2.2: Latency of preemptible function interface

preemptible function’s stack. This in turn would cause the language runtime to unwind each stack frame, invoking local variables’ destructors in the process.

2.3 Evaluation

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.

2.3.1 Microbenchmarks

Table 2.2 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.

2.3.2 Image decompression

Unlike state of the art approaches, lightweight preemptible functions support 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 [36], 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 5.1. 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
Figure 2.3: *libpng* in-memory decode times for two different images

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 2.3a). 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 2.3b). Without asynchronous cancelability, the pthreads approach was unable to interrupt the thread. Here, launch() exceeded the deadline by just 100 µs, a figure that includes deviation due to the 100-µs preemption interval in addition to *libinger’s* own overhead.
It again had the lowest variance.
Chapter 3

Nonreentrancy and selective relinking

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 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 2.2.1). 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 2.2.2). Listing 3.1 shows the impact on an example program.

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.

### 3.0.1 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. 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 [13].

It may seem like namespaces provide the isolation we need: whenever we `launch(F)`, we
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 3.1: Demo of isolated (1) vs. shared (2&3) state

typedef long libset_t;

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

Listing 3.2: libgotcha C interface

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 3.1 would fail assertion (2).

3.0.2 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 3.2

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 3.2 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.

### 3.0.3 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. We work around this by rebuilding glibc with an increased `TLS_STATIC_SURPLUS`. It is useful to also raise the maximum number of namespaces by increasing `DL_NNS`.

### 3.0.4 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.\(^1\)

Selective relinking works by shadowing the GOT. 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.

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

---

\(^1\)Some other functions are instead resolved lazily at their first invocation.

\(^2\)Hence the name *libGOTcha*.
register of its address calculation[3] 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[4].

3.0.5 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 5.1 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 2.2.4 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 callback that it uses to reenable preemption as soon as any uncopyable function returns.

3.0.6 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:

---

[3] 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.

[4] This does not break applications with existing segfault handlers: we intercept their calls to sigaction(), and forward the signal along to their handler whenever we are unable to resolve an address ourselves.
### Symbol resolution scheme

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Time without <em>libgotcha</em> (ns)</th>
<th>Time with <em>libgotcha</em> (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eager (load time)</td>
<td>2 ± 0</td>
<td>14 ± 0</td>
</tr>
<tr>
<td>lazy (runtime)</td>
<td>100 ± 1</td>
<td>125 ± 0</td>
</tr>
<tr>
<td>global variable</td>
<td>0 ± 0</td>
<td>3438 ± 13</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Baseline</th>
<th>Time without <em>libgotcha</em> (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gettimeofday()</td>
<td></td>
<td>19 ± 0</td>
</tr>
<tr>
<td>getpid()</td>
<td></td>
<td>44 ± 0</td>
</tr>
</tbody>
</table>

(b) Library functions and syscalls without *libgotcha*

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Time with <em>libgotcha</em> (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncopyable call</td>
<td>21 ± 0</td>
</tr>
<tr>
<td>Uncopyable call + callback</td>
<td>25 ± 0</td>
</tr>
</tbody>
</table>

(c) Uncopyable calls triggering a libset switch

Table 3.1: Runtime overheads of accessing dynamic symbols

- 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.

### 3.1 Evaluation

Recall that linking an application against *libgotcha* imposes additional overhead on most dynamic symbol accesses; we report these overheads in Table 3.1. Eager function calls account for almost all of a modern program’s dynamic symbol accesses: Direct access to global variables is rarer now that thread safety is a pervasive concern, with libc removing the `errno` symbol in favor of an `_errno_location()` helper function [31], and compilers now translating thread-local variable accesses into calls to `_tls_get_addr()` or offsets from a segment register [15]. As explained in Section 3, lazy resolution only occurs the first time an object file calls a particular function.

Table shows that the *libgotcha* eager function call overhead of 14 ns is on par with the cost of a trivial C library function and one-third that of a simple system call. 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 3. Table 3.1c breaks this
down to show the cost of notification callbacks at the conclusion of such a call (always required by *libinger*).

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.
Shinjuku observes that “there have been several efforts to implement efficient, user-space thread libraries. They all focus on cooperative scheduling” [26]. We agree that such libraries are rare, and attribute this to a lack of natural abstractions to support them. (Though RT from Section 2.1 could be a counterexample, its lack of nonreentrancy support makes it far from general purpose.)

Although the preemptible function is a fundamentally synchronous abstraction, its simplicity makes it readily composable, and indeed, well-suited to implementing preemptive threading. As a proof of concept, we have created a preemptively-scheduled userland thread library, libturquoise¹, by modifying the tokio-threadpool [19] work-stealing scheduler from the Rust futures ecosystem.

To migrate the thread pool workers to preemptive scheduling, we made them poll each task future from within a preemptible function. We did this in just 120 new lines of Rust, 50 of them added to version 0.1.16 of the thread library and 70 spent augmenting libinger’s Rust API with a reusable preemptible futures adapter.

Currently, libturquois 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.

### 4.0.1 Preemptible futures

For seamless interoperation between preemptible functions and the futures ecosystem, we built a preemptible future adapter that wraps the libinger API. This can be used to pass preemptible functions into a platform designed to process futures.

Because of languages’ differing futures, this integration is not portable like the core API. Fortunately, its implementation is a straightforward application of `pause()` to propagate cooperative yields across the preemptive function boundary: we present the general construction of the preemptible future type and an algorithm for polling one in Listing 4.1

¹ so called because it implements “green threading with a twist”
function PreemptibleFuture(Future fut,  
        Num timeout):
    function adapt():
        while poll(fut) == NotReady:
            pause()
        fut.linger = launch(adapt, 0)
        fut.timeout = timeout  
    return fut

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 4.1: Futures adapter type (pseudocode)

4.1 Evaluation

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 [23]. We built hyper against libturquoise configured 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 [20] 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, an 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 4.1 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 the preemptible functions stack keeps the tail latency of short requests scaling linearly at the cost of a modest 4.5% median latency overhead when not
under attack.

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.
Chapter 5

Microsecond-scale microservices

This chapter provides a case study of how lightweight preemptible functions could be used to create the serverless platform of the future.

Modern cloud computing environments strive to provide users with fine-grained scheduling and accounting, as well as seamless scalability. The most recent face to this trend is the “serverless” model, in which individual functions, or microservices, are executed on demand. Popular implementations of this model, however, operate at a relatively coarse granularity, occupying resources for minutes at a time and requiring hundreds of milliseconds for a cold launch. In this paper, we describe a novel design for providing “functions as a service” (FaaS) that attempts to be truly micro: cold launch times in microseconds that enable even finer-grained resource accounting and support latency-critical applications. Our proposal is to eschew much of the traditional serverless infrastructure in favor of language-based isolation. The result is microsecond-granularity launch latency, and microsecond-scale preemptive scheduling using high-precision timers.

5.1 Introduction

As the scope and scale of Internet services continues to grow, system designers have sought platforms that simplify scaling and deployment. Services that outgrew self-hosted servers moved to datacenter racks, then eventually to virtualized cloud hosting environments. However, this model only partially delivered two related benefits:

1. Pay for only what you use at very fine granularity
2. Scale up rapidly on demand

The VM approach suffered from relatively coarse granularity: Its atomic compute unit of machines were billed at a minimum of minutes to months. Relatively long startup times often required system designers to keep some spare capacity online to handle load spikes.

These shortcomings led cloud providers to introduce a new model, known as serverless computing, in which the customer provides only their code, without having to configure its environment. Such “function as a service” (FaaS) platforms are now available as AWS Lambda [4], Google Cloud Functions [21], Azure Functions [33], and Apache OpenWhisk [6]. These platforms provide a model in which: (1) user code is invoked whenever some event occurs (e.g., an HTTP API
request), runs to completion, and nominally stops running (and being billed) after it completes; and (2) there is no state preserved between separate invocations of the user code. Property (2) enables easy auto-scaling of the function as load changes.

Because these services execute within a cloud provider’s infrastructure, they benefit from low-latency access to other cloud services. In fact, acting as an access-control proxy is a recurring microservice pattern: receive an API request from a user, validate it, then access a backend storage service (e.g., S3) using the service’s credentials.

In this paper, we explore a design intended to reduce the tension between two of the desiderata for cloud functions: low latency invocation and low cost. Contemporary invocation techniques exhibit high latency with a large tail; this is unsuitable for many modern distributed systems which involve high-fanout communication, sometimes performing thousands of lookups to handle each user request. Because user-visible response time often depends on the tail latency of the slowest chain of dependent responses [12], shrinking the tail is crucial [24, 46, 30, 25].

Thus we seek to reduce the invocation latency and improve predictability, a goal supported by the impressively low network latencies available in modern datacenters. For example, it now takes < 20µs to perform an RPC between two machines in Microsoft Azure’s virtual machines [18]. We believe, however, that fully leveraging this improving network performance will require reducing microservices’ invocation latencies to the point where the network is once again the bottleneck.

We further hypothesize—admittedly without much proof for this chicken-and-egg scenario—that substantially reducing both the latency and cost of running intermittently-used services will enable new classes and scales of applications for cloud functions, and in the remainder of this paper, present a design that achieves this. As Lampson noted, there is power in making systems “fast rather than general or powerful” [28], because fast building blocks can be used more widely.

Of course, a microservice is only as fast as the slowest service it relies on; however, recall that many such services are offered in the same clouds and datacenters as serverless platforms. Decreasing network latencies will push these services to respond faster as well, and new stable storage technologies such as 3D XPoint (projected to offer sub-microsecond reads and writes) will further accelerate this trend by offering lower-latency storage.

In this paper, we propose a restructuring of the serverless model centered around low-latency: lightweight microservices run in shared processes and are isolated primarily with language-based compile-time guarantees and fine-grained preemption.

5.2 Motivation

Our decision to use language-based isolation is based on two experimental findings: (1) Process-level isolation is too slow for microsecond-scale user functions. (2) Commodity CPUs support task preemption at microsecond scale. We conducted our experiments on an Intel® Xeon® E5-2683 v4 server (16 cores, 2.1 GHz) and Linux 4.13.0. [1]

[1] Source code for the benchmarks in this paper is available from https://github.com/efficient/microservices_microbebenchmarks

24
5.2.1 Process-level isolation is too slow

We use a single-machine experiment to evaluate the invocation overhead of different isolation mechanisms: Microservices run on 14 worker CPU cores. Another core runs a dispatcher process that launches microservices on the workers. All requests originate at the dispatcher (which in a full serverless platform would forward from a cluster scheduler); it schedules $\leq 14$ microservices at a time, one per worker core, choosing from a pool of 5,000.

To provide a comparison against contemporary system designs, we use two different isolation mechanisms:

1. **Process-based isolation**: Each microservice is a separate process. We expect this approach to exhibit latency at least as low as the container isolation common in contemporary serverless deployments.

2. **Language-based isolation**: Each worker core hosts a single-threaded worker process that directly executes different microservices, one at a time. In this approach, shown in Figure 5.1, a worker process runs a microservice by calling its registered function; we assume that the microservice function can be isolated from the worker process with language-based isolation techniques that we discuss in Section 5.3. The dispatcher schedules microservices on worker processes by sending them requests on a shared memory queue, which idle worker processes poll.

We use 5,000 copies of a Rust microservice that simply records a timestamp: latency is measured between when the dispatcher invokes a microservice and the time that microservice records. There are two experiment modes:

**Warm-start requests.** We first model a situation where all of the microservices are already resident on the compute node. In the case of process-based isolation, the dispatcher launches all 5,000 microservices at the beginning of the experiment, but they all block on an IPC call; the dispatcher then invokes each microservice by waking up its process using a UDP datagram. In the case of language-based isolation, the microservices are dynamic libraries preloaded into the worker processes.

Table 5.1 shows the latency and throughput of the two methods. We find that the process-based isolation approach takes 9 $\mu$s and achieves only 300,000 warm microservice invocations.
per second. In contrast, language-based isolation achieves 1.2 $\mu$s latency (with a tail of just 2.0 $\mu$s) and over 5 million invocations per second.

Considering that the FaRM distributed computing platform achieved mean TATP transaction commit latencies as low as 19 $\mu$s in 2015 \cite{14}, a 9 $\mu$s microservice invocation delay represents almost 50% overhead for a microservice providing a thin API gateway to such a backend. We therefore conclude that even in the average case, process-based isolation is too slow for microsecond-scale scheduling. Furthermore, IPC overhead limits invocation throughput.

Process-based isolation also has a higher memory footprint: loading the 5,000 trivial microservices consumes 2 GiB of memory with the process-based approach, but only 1.3 GiB with the language-based one. However, this benefit may reduce as microservices’ code sizes increase.

Cold-start requests. Achieving ideal wakeup times is possible only when the microservices are already resident, but the tail latency of the serverless platform depends on those requests whose microservices must be loaded before they can be invoked. To assess the difference between process-based and language-based isolation in this context, we run the experiment with the following change: In the former case, the dispatcher now launches a transient microservice process for each request by \texttt{fork()/exec()}’ing. In the latter, the dispatcher asks a worker to load a microservice’s dynamic library (and unload it afterward). The results in Table 5.1 reveal an order-of-magnitude slip in the language-based approach’s latency; however, this is overshadowed by the three orders of magnitude increase for process-based isolation.

5.2.2 Intra-process preemption is fast

In a complete serverless platform, some cluster-level scheduler would route incoming requests to individual worker nodes. Since we run user-provided microservices directly in worker processes, a rogue long-running microservice could thwart such scheduling by unexpectedly consuming the resources of a worker that already had numerous other requests queued. We hypothesize that, in such situations, it is better for tail latency to preempt the long microservice than retarget the waiting jobs to other nodes in real time. (Only the compute node already assigned a request is well positioned to know whether that request is being excessively delayed: whereas other nodes can only tell that the request hasn’t yet completed, this node alone knows whether it has been scheduled.) At our scale, this means a preemption interval up to two orders of magnitude faster than Linux’s default 4 ms process scheduling quantum.

Fortunately, we find that high-precision event timers (HPETs) on modern CPUs are sufficient
for this task. We measure the granularity and reliability of these timers as follows: We install a
signal handler and configure a POSIX timer to trigger it every $T \mu s$. Ideally, this handler would
always be called exactly $T \mu s$ after its last invocation; we measure the deviation from $T$ over
65,535 iterations. We find that the variance is smaller than 0.5 $\mu s$ for $T \geq 3 \mu s$. This shows that
intra-process preemption is fast and reliable enough for our needs.

5.3 Providing Isolation

Consolidating multiple users’ jobs into a single process requires addressing security and isolation.
We aim to do it without compromising our ambitious performance goals.

Our guiding philosophy for achieving this is “language-based isolation with defense in depth.”
We draw inspiration from two recently-published systems whose own demanding performance
requirements drove them to perform similar coalescing of traditionally independent components:
NetBricks [35] is a network functions runtime for providing programmable network capabilities;
it is unique among this class of systems for running third-party network functions in-process
rather than in VMs. Tock [29] is an embedded microkernel whose servers (“capsules”) form a
common compilation unit and communicate using type-safe function calls. As their primary
defense against untrusted code, both systems leverage Rust [3], a new type-safe systems pro-
gramming language.

Rust is a strongly-typed, compiled language that uses a lightweight runtime similar to C. Un-
like many other modern systems languages, Rust is an attractive choice for predictable perfor-
mance because it does not use a garbage collector. It provides strong memory safety guarantees
by focusing on “zero-cost abstractions” (i.e., those that can be compiled down to code whose
safety is assured without runtime checks). In particular, safe Rust code is guaranteed to be free of
null or dangling pointer dereferences, invalid variable values (e.g., casts are checked and unions
are tagged), reads from uninitialized memory, mutations of non-mut data (roughly the equivalent
of C’s const), and data races, among other misbehaviors [42].

We require each microservice to be written in Rust (although, in the future, it might be possi-
ble to support subsets of other languages by compiling them to safe Rust), giving us many aspects
of the isolation we need. It is difficult for microservices to crash the worker process, since most
segmentation faults are prevented, and runtime errors such as integer overflow generate Rust
panics that we can catch. Microservices cannot get references to data that does not belong to
them thanks to the variable and pointer initialization rules.

Given our performance goals, there is a crucial isolation aspect that Rust does not provide:
there is nothing to stop users from monopolizing the CPU. Our system, however, must be preempt-
ive. We are unaware of existing preemption techniques that work at microsecond scales. Note
that coroutine-like cooperative multitasking approaches (such as lightweight threads in Go [2]
and Erlang [1]) are not preemptive, so they do not work for us. We briefly discuss our solution
to this in the following section; it depends on installing a SIGALRM handler and ensuring that
trusted code within the process handles the signal.

Our defense-in-depth comes from using lightweight operating-system protections to block
access to certain system calls, as well as the proposed mechanisms in Section 5.6. Some system
calls must be blocked to have any defense at all; otherwise, the microservice could create kernel
threads (e.g., fork()), create competition between threads (e.g., nice()), or even terminate the
entire worker (e.g., `exit()`). Finally, user functions should not have unmonitored file system access.

We propose to block system calls using Linux’s `seccomp()` system call \cite{seccomp}; each worker process should call this during initialization to limit itself to a whitelisted set of system calls. Prior to lockdown, the worker process should install a SIGSYS handler for regaining control from any microservice that attempts to violate the policy.

## 5.4 Providing Preemption

The system must be able to detect and recover from microservices that, whether maliciously or negligently, attempt to run for longer than permitted. The two parts of this problem are (1) regaining control of the CPU and (2) aborting and cleaning up after the user code.

As proposed in Section \ref{sec:preemption}, regaining control of the CPU happens when a signal (SIGALRM) from the kernel transfers control to the worker process’s handler. The handler then checks how long the current microservice has been running and decides whether it should be killed. (We register the handler using the `SA_RESTART` flag to `sigaction()` so that any interrupted blocking syscalls are restarted transparently.)

**How often should the handler execute (the quantum)?** We showed in Section \ref{sec:preemption} that microsecond-scale preemption is achievable, but can it be done efficiently? To find out, we wrote a microservice that measures the throughput of computing SHA-512 hashes over 64 B of data at a time. We then subjected its worker process to SIGALRMs, varying the quantum and observing the resulting hashing throughput. Figure \ref{fig:quantum} illustrates that by a quantum of about 20 µs, throughput had reached around 90% of baseline. Considering this performance degradation, acceptable we adopt this quantum and prescribe a runtime budget of 113 - 20 = 93 µs so that we can kill over-budget microservices in time to avoid violating our tail latency SLO.

**How do we clean up a terminated microservice?** We now discuss our mechanism for aborting and cleaning up after a microservice exceeds its runtime budget. POSIX signal handlers receive as an argument a pointer to their context, a snapshot of the process’s PCB (process control block) at the moment before it received the signal. When the handler returns, the system will transfer control back to the point described by the context, so a naïve way for our worker processes to regain control would be to reset its GPRs (general-purpose registers) to values recorded just before the worker’s tight scheduling loop. This approach, however, would not release the microservice’s state or memory allocations back to the worker.

One of the few heavyweight components of the Rust runtime is panic handling, reminiscent of C++’s exception handling. The compiler inserts landing pads into each function that call the destructors for the variables in its stack frame: if the program ever panics, the standard library uses these to unwind the stack. We co-opt this functionality by having the SIGALRM handler set its context to raise an explicit panic in a fake stack frame just above the real top of the stack. Section \ref{sec:panic} discusses the limitations and security ramifications of this approach.

---

\footnote{For defense in depth, the worker process should be prevented from subsequently modifying this signal-handling configuration.}
Figure 5.2: Effect of SIGALRM quantum on hashing tput.
5.5 Deployment

We now describe our microservices in the broader context of our full proposed serverless system. We clarify their lifecycle, interactions with the compute nodes, and the trust model for the cloud provider.

Users submit their microservices in the form of Rust source code, allowing the serverless operator to pass the -Funsafe-code compilation flag to reject any unsafe code. This process need not occur on the compute nodes, provided the deployment server tasked with compilation runs the same version of the Rust compiler. The operator needs to trust the compiler, standard library, and any libraries against which it will permit the microservice to link (since they might contain unsafe code), but importantly need not worry about the microservice itself.

We believe that restricting microservices to a specific list of permitted dependencies is reasonable. Any library that contains only safe Rust code could be whitelisted without review. To approximate the size of such a list given the current Rust ecosystem, we turn to a 2017 study by the Tock authors that found just under half of the Rust package manager’s top 1000 most-downloaded libraries to be free of unsafe code. They caution that many of those packages have unsafe dependencies, but reviewing a relatively small number of popular libraries would open up the majority of the most popular packages.

If the application compiles (is proven memory-safe) and links (depends only on trusted libraries) successfully, the deployment server produces a shared object file, which the provider then distributes to each compute node on which it might run. Then, in order to ensure that invokers will experience the warm-start latencies discussed in Section 5.2, those nodes’ dispatcher processes should instruct one or more of their workers to preload the dynamic library. If the provider experiences too many active microservices for its available resources, it can unload some libraries; on their next invocation, they will experience higher (cold start) invocation latencies as they synchronously load the dynamic library.

5.6 Future Work

As noted above, our exploration is preliminary; this section outlines several open questions. These questions fall into two categories: shortcomings in our current implementation and defense-in-depth safeguards against unexpected failures (e.g., compiler bug or the operator allowing use of a buggy or malicious library).

Non-reentrancy. Our use of Rust panics to unwind the stack during preemption can corrupt the internal state of non-reentrant functions (e.g., Rust’s dynamic allocator). Possible fixes include blacklisting these functions and delaying preemption until they are finished or replacing the problematic function with a safe one (e.g., a custom memory allocator).

Host process. Our current implementation does not provide isolation between the dispatcher and worker processes. We plan to apply standard OS techniques to reduce the chance of interference by a misbehaving worker. Examples include auditing interactions with the shared memory.
region to ensure invalid or inconsistent data originating from a worker cannot create an unrecoverable dispatcher error; handling the SIGCHLD signal to detect a worker that has somehow crashed; and keeping a recovery log in the dispatcher process so that any user jobs lost to a failed worker process can be reassigned to operational workers.

**Further defense in depth with ERIM.** ERIM outlines a set of techniques and binary rewriting tools useful for using Intel’s Memory Protection Keys to restrict memory access by threads within a process [44]. While preliminary and without source yet available, this appears to be an attractive approach for defense-in-depth both within worker processes and between the workers and the dispatcher.

**Library functions.** As with system calls, there may exist library functions in Rust (and certainly in libc, which we deny by default) that are unsafe for microservices to access. Because the Rust standard library requires unsafe code, defense-in-depth suggests that a whitelisting-based approach should be employed for access to its functions. Certainly library functions must be masked—for example, our use of Rust’s panic handler for preemption means that we must deny microservice code the ability to catch the panic and return to execution. Although we mitigate this possibility by detecting and blacklisting microservices that fail to yield under a SIGALRM, it would be desirable to block such behavior entirely. Possible options include using the dynamic linker to interpose stub implementations or linking against a custom build of the library, or using more in-depth static analysis.

**Resource leaks.** Safe Rust code provides memory safety, but it cannot prevent memory leaks [43]. For example, destructor invocation is not guaranteed using Rust’s default reference counting-based reclamation; therefore, unwinding the stack during preemption is not guaranteed to free all of a microservice’s memory or other resources. Potential solutions are interposing on the dynamic allocator to record tracking information (likely proving expensive) or using per-microservice heaps that main worker process can simply deallocate when terminating a microservice. The worker can also deallocate other resources, such as unclosed file descriptors. If these checks end up being too expensive, the worker could execute its cleanup after a certain number of microservices have run or when the load is sufficiently low.

**Side channels.** Our current approach is vulnerable to side-channel attacks [32, 27]. For example, microservices have access to the memory addresses and timings of dynamic memory allocations, as well as the numbers of opened file descriptors. Although side-channels exist in many systems, the short duration of microservice functions may make mounting such attacks more challenging; nevertheless, standard preventative practices found in the literature should apply. Despite the security challenges of running microservice as functions, worker processes are still well-isolated from the rest of the system. Worst case, the central dispatcher process can restart a failed worker and automatically ban suspect microservices.
5.7 Conclusion

In order to permit applications to fully leverage the 10s of $\mu$s latencies available from the latest datacenter networks, we propose a novel design for serverless platforms that runs user-submitted microservices within shared processes. This structure is possible because of language-based compile-time memory safety guarantees and microsecond-scale preemption. Our implementation and experiments demonstrate that these goals of high throughput, low invocation latency, and rapid preemption are achievable on today’s commodity systems, while potentially supporting hundreds of thousands of concurrently available microservices on each compute node. We believe that these two building blocks will enable new FaaS platforms that can deliver single-digit microsecond invocation latencies for lightweight, short-lived tasks.
Chapter 6

Proposed work

We presented the lightweight preemptible function, a composable new abstraction for invoking a function with a timeout. The implementation, `libinger`, serves as the foundation of a first-in-class preemptive userland thread library. Our evaluation shows that lightweight preemptible functions have overheads of a few percent (lower than similar OS primitives), yet enable new functionality.

6.1 Remaining work

I propose extending the work already completed in all three of the following ways:

Cancellation resource cleanup for the Rust interface Although we currently support asynchronous cancellation of timed-out preemptible functions via `libinger`’s `cancel()` facility, we do not yet perform any automatic cleanup of already-allocated resources. This is impossible in general for C programs because the language lacks a destructor mechanism; however, I intend to add at least partial support for doing so for Rust programs. The basic principle will be to throw an artificial exception (Rust panic) on the preemptible function’s execution stack, thereby unwinding the stack and invoking local variables’ destructors. This approach alone will not guarantee comprehensive cleanup, because such variables may not have been fully initialized at the time of preemption; for the same reason, it may have safety ramifications that will merit additional study. I have some ideas about how to augment the technique, such as by extending `libgotcha` to keep a running cache of the several most recent allocations and deallocations of common resources (e.g., memory and file descriptors) from each libset. While exhaustive cleanup may prove difficult to achieve, I hope to at least catalogue situations where we guarantee not to leak certain classes of resource, and perhaps provide a tunable allowing users to request full resource tracking if needed.

Automatic selection and variation of timer frequency For simplicity, the current `libinger` implementation subscribes to timer signals spaced a globally constant interval apart throughout the entire duration of each preemptible function. To improve efficiency while preserving preemption granularity, I plan to dynamically determine this interval based on the requested timeout. For long-running functions, this will include delaying the first of these signals until shortly before the timeout would expire. Maintaining accuracy across multiple CPUs will probably require
building in a calibration routine to infer configuration parameters.

**Support OpenSSL and benchmark hyper with HTTPS** Earlier in *libgotcha*'s development history, it was able to run nginx with OpenSSL. Recent attempts at getting hyper to run with OpenSSL have ended in crashes, yet this configuration is desirable for measurement because it exhibits a far greater number of dynamic function calls, of which *libgotcha* alters the performance characteristics. I aim to support and benchmark the configuration, which will likely involve regression testing using the old nginx setup.

Additionally, I will complete one of the following projects, as selected by the committee:

**Optimize libset reinitialization** The high cost of cancellation is a consequence of our libset reinitialization approach, which currently involves unloading all libraries from the libset, then reloading them and running all their constructors in the process. I believe cheaper schemes are possible, such as checkpointing only the writeable portions of each library’s address space (e.g., by replacing their page mappings with copy-on-write pages backed by their saved contents immediately after return from the library’s constructors).

**Achieve even finer-grained preemption** The *Shinjuku* authors included IPI microbenchmarks that suggest it should be possible for us to achieve interrupt latencies and spacing within a small constant factor of theirs. I could attempt to achieve this by reusing some of their optimizations to POSIX contexts and adding a specialized timer signal delivery mechanism to the kernel to reduce the number of ISR instructions at the expense of generality.

**Implement “libingerOS” container framework** Generalizing the serverless platform case study, it should be possible to generate a utility that takes two or more position-independent executables and combines them into a single process whose thread(s) are timeshared between the “programs” using preemptible functions. Obviously, this would only provide memory isolation for programs implemented in memory-safe languages. This would be implemented on top of *libinger* as a sample application.

**Implement a caching RPC framework** The preemptible functions API naturally lends itself to problems of caching partial computations, and one interesting case study would be a caching RPC framework. I imagine this working as follows: Clients would enclose alongside each request the timeout they were using when listening. The server would process each request inside its own preemptible function. When the client timed out waiting for a response, the server would simultaneously time out and cache the continuation. Subsequent requests for the same procedure with identical inputs would resume the interrupted computation from where it left off.

### 6.2 Timeline

- 24 April 2020: ATC ’20 author notification
- 4 June 2020: ATC ’20 cameraready deadline (if applicable)
• June 2020: Start cancellation resource cleanup, fix OpenSSL support
• July 2020: Continue cancellation resource cleanup, fix OpenSSL support
• August 2020: Finish cancellation resource cleanup
• September 2020: Speaking skills talk, start automatic preemption intervals
• October 2020: Finish and evaluate automatic preemption intervals
• November 2020: Start committee-selected project
• December 2020: Finish and evaluate committee-selected project
• January–March 2021: Thesis writing, run any final experiments
• April 2021: Finish thesis
• May 2021: Defense
Bibliography


[38] seccomp(). seccomp(2) manual page from Linux man-pages project, Nov. 2017.


