ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH zürich

Bachelor's Thesis Nr. 605b

Systems Group, Department of Computer Science, ETH Zurich

Porting Dandelion to the Lauberhorn smart NIC

by

Sven Glinz

Supervised by

Pengcheng Xu, Prof. Dr. Timothy Roscoe

February 2026–August 2026

D INFK

1 Abstract

As datacenter applications are decomposed into microservices and serverless functions, an increasing fraction of end-to-end latency goes toward receiving and dispatching small RPC requests, rather than compute. Lauberhorn, a smart NIC, addresses this overhead by participating in the CPU's cache-coherence protocol, scheduling requests directly onto cores without kernel involvement or polling on the dispatch path. This thesis extends the Lauberhorn runtime to support non-trivial RPC applications, most notably handlers that issue RPC calls of their own. To demonstrate the result, we port the core of Dandelion, a serverless computing platform, onto Lauberhorn. Dandelion's software scheduler is removed entirely, and serverless compositions execute as graphs of nested calls dispatched by Lauberhorn. As Lauberhorn hardware was not yet fully functional, all development and evaluation builds on a kernel module that mocks Lauberhorn's behavior in software. Single Dandelion functions dispatched through Lauberhorn see end-to-end latency drop by about 9%. Compositions instead incur a small penalty that grows with depth, as without hardware each nested call is routed through the kernel module. A latency-throughput evaluation for a simple handler shows the runtime sustaining over a hundred thousand requests per second. On real hardware, which removes the kernel module from the critical path, throughput should climb further. These results indicate that Lauberhorn together with the runtime provides a promising platform for fast, low-overhead RPC dispatch.

Acknowledgements

I would like to thank my supervisor Pengcheng Xu for his continuous support with the project, his constructive inputs and his willingness to occasionally extend our allocated weekly time for further discussions which were always very insightful. I also want to thank Prof. Dr. Roscoe for making this project possible, and Tom Kuchler for his introduction into the Dandelion runtime. Finally, I thank my family, friends, and my girlfriend for their support during the last semester.

Contents

2 Introduction

Datacenter applications are increasingly structured as microservices. Rather than a monolithic application whose components communicate through function calls, the application is decomposed into fine-grained modules that communicate through Remote Procedure Calls (RPCs) over the network [10]. Serverless computing further pushes this decomposition with Functions as a Service (FaaS), in which stateless functions are invoked in isolated environments in response to events. As the size and compute cost of requests decrease, communication accounts for a growing share of request latency [8]. Beyond network propagation delay, a significant part of this is end-system latency. The server must unmarshal, demultiplex, and dispatch each RPC before a handler ever runs. This overhead can quickly dominate actual work for microsecond-scale RPCs.

Kernel-bypass solutions shorten the path between packet arrival and handler invocation, but typically trade energy efficiency for latency. Usually, cores spin-wait on packet queues, burning cycles whether or not requests arrive [14]. Lauberhorn avoids this tradeoff by connecting the NIC and CPU over a cache-coherent interconnect, using loads and stores on NIC-homed cache lines for communication between the CPU and the NIC [12]. This enables low-overhead RPC dispatch without dedicating spinning cores to the NIC. Moreover, coherence protocol messages give the NIC visibility into per-core state, allowing it to dispatch each incoming RPC directly to a suitable core.

Applications interact with Lauberhorn through a C runtime that registers RPC handlers and starts worker threads. As requests arrive, Lauberhorn schedules these worker threads and the runtime executes the handler associated with the request.

The original runtime supports only self-contained handlers. However, many RPC services are built around nested invocation, where handlers issue RPCs of their own to downstream services [10]. This thesis extends the Lauberhorn runtime to support such non-trivial applications, with nested RPC calls as the central addition. To demonstrate the extended runtime on a realistic workload, we modify a part of Dandelion [6], a serverless platform, to invoke functions via Lauberhorn. By delegating scheduling to Lauberhorn, Dandelion's own scheduler can be bypassed entirely. As the Lauberhorn hardware was not fully functional at the start of this thesis, we additionally contribute a kernel module that simulates Lauberhorn's dispatch and scheduling behavior in software and serves as an evaluation platform.

The remainder of this thesis is structured as follows. Section 3 provides background on Lauberhorn, its accompanying software, and Dandelion. Section 4 describes the kernel module that simulates Lauberhorn in software. Section 5 presents the runtime extensions, in particular support for nested RPCs, and Section 6 presents the port of Dandelion onto the extended runtime. Sections 7 and 8 evaluate the resulting system and its scalability limits. Finally, Section 9 compares the runtime to other works before Section 10 concludes.

3 Background

3.1 Lauberhorn

The Lauberhorn smart network interface card (NIC) exploits cache-coherent interconnects such as ECI (Enzian Coherence Interface) on the Enzian [11] to interact more directly with the CPU than a conventional NIC, allowing it to dispatch RPC workloads faster than existing kernel-bypass solutions. The following gives a brief overview of Lauberhorn's core functionality. A detailed description, including how the communication via ECI works, is provided by Xu and Roscoe [14].

Because the NIC participates in the CPU's cache-coherence protocol, the NIC and the cores can exchange data through loads and stores on NIC-homed cache lines, with no kernel involvement on the fast path via the 2F2F protocol [12]. This is particularly fast and efficient compared to traditional DMA-based data transfers for small messages up to a few KiB [14].

Lauberhorn builds an RPC dispatch mechanism on top of this. A worker thread requests its next RPC by issuing a load on a designated cache line, and the NIC delivers work by fulfilling that load. Until a request arrives, the NIC can stall the load. As such a load cannot block indefinitely without triggering a "bus error", it completes empty after a few milliseconds and control returns to the worker thread [14], which may perform periodic housekeeping (e.g., reclaiming state of stale requests) before re-issuing the load. Waiting for work is therefore a loop of cache line loads rather than busy waiting, consuming less energy.

To interact with the NIC, Lauberhorn provides two software components. A *kernel module* configures the hardware and exposes it to the system - as a regular network device, and additionally as a character device through which applications register themselves and their RPC endpoints (described in Section 4). A *user-space runtime* builds on this interface, offering an API for registering an application as an RPC endpoint, registering individual RPC handlers, and spawning worker threads that exchange data directly with the NIC via the described protocol.

Data Path

On Lauberhorn, delivering and transmitting RPC payloads (i.e. the fast data path) involves only the NIC and the runtime. On packet arrival, Lauberhorn parses just enough to extract the RPC header needed to route the request to a particular worker core. If a worker thread of the destination service is currently stalled on a cache line load, Lauberhorn fulfills the request immediately by writing the arguments, together with a pointer to endpoint metadata, such as the associated handler, directly into the cache line, and the worker jumps to the handler without any OS involvement. Payloads larger than a single cache line are provided through additional NIC-homed *overflow* cache lines, which the worker reads once the initial load has signaled a complete request. Currently, the argument data is handed over still serialized. Deserialization is performed by the runtime before invoking the handler, though hardware-accelerated de-

serialization is planned.

If a worker is stalled, but the worker is not running the specific process that registered the handler, Lauberhorn instead raises an interrupt, and the OS schedules an appropriate thread onto the core. If no core is available for scheduling work, Lauberhorn queues the packet and waits for the next cache line load. As Lauberhorn focuses on short-lived workloads, workers are never interrupted and run all handlers to completion. The worker eventually writes its reply through the same cache line mechanism and the NIC places it on the wire.

Packets that do not match any registered RPC endpoint are forwarded via the same cache line mechanism, to a dedicated *bypass core*. This core runs a kernel thread that injects the packets into the kernel's network stack. Lauberhorn thus remains usable as a conventional NIC for non-RPC traffic.

As Lauberhorn infers the current scheduling state from the cache-coherence traffic itself and places requests onto cores autonomously, the OS scheduler must not interfere with these placement decisions. All worker cores are therefore fully isolated from the Linux scheduler, interrupt handling, and RCU callbacks (via `isolcpus`, `nohz_full`, and `rcu_nocbs`), and run only the worker threads managed by Lauberhorn.

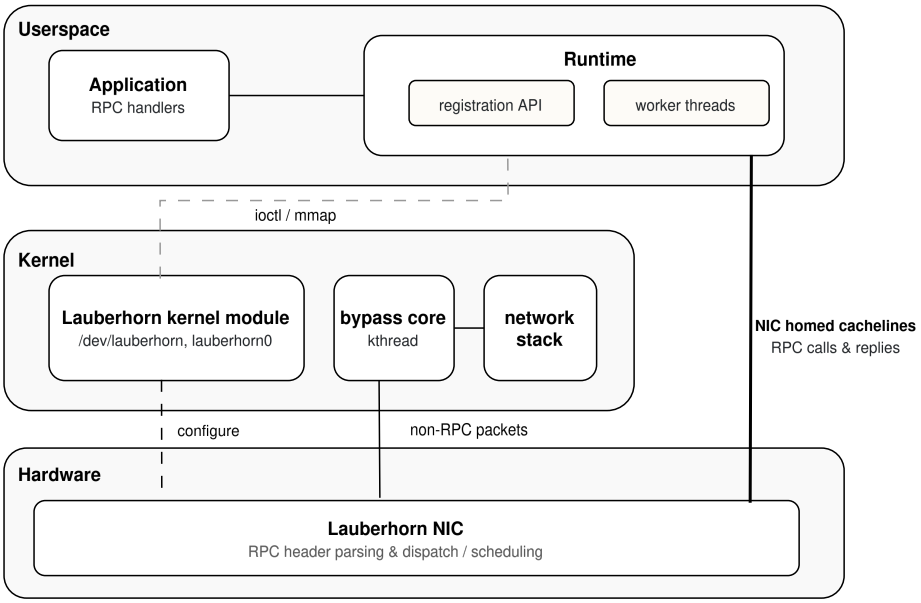

Figure 1: Lauberhorn Components.

3.2 ONC RPC

To understand how RPC services are represented, this section gives a brief overview of ONC RPC, the protocol on which Lauberhorn's RPC support is built.

ONC RPC (Open Network Computing Remote Procedure Call) was developed by Sun Microsystems in the 1980s and remains notably in use today as the RPC protocol underlying the Network File System (NFS) [3]. This section describes version 2 of the protocol [13], the version Lauberhorn uses.

ONC RPC supports both UDP and TCP transport. Lauberhorn currently uses UDP exclusively. Since neither retransmission nor framing of application data across multiple UDP packets is supported, any handler registered through the runtime is limited to at most 1500 bytes of data, matching the standard Ethernet MTU. Likewise, Lauberhorn currently does not support any of the ONC RPC authentication schemes. Every call is treated as `AUTH_NONE`.

XDR: External Data Representation

Machines exchanging RPC data may differ in endianness, word size, and alignment. Therefore, ONC RPC describes all data in XDR, an architecture-independent data description language [2]. An XDR definition resembles a C struct declaration (Listing 1), but specifies only the data's logical layout. The wire encoding is fixed by the XDR standard, rather than declared explicitly. Scalars such as integers are always encoded big-endian on the wire, regardless of the host's native byte order. Beyond scalars, XDR supports nested structs and arrays. A fixed-size array is encoded as the concatenation of its elements, while a variable-length array (or a variable-length opaque byte sequence) is preceded on the wire by its length as an unsigned 4-byte big-endian integer, with the encoded data padded to a 4-byte boundary.

```
struct add_call {
    int a;
    int b;
};

struct add_resp {
    int sum;
};
```

Listing 1: XDR definitions of an argument and a result struct.

RPC Service

An RPC service is described as a *program* which groups related *procedures* under one or more *versions*. Listing 2 defines a program `MATH_PROG` with program number `0x12345`, containing a single version (1) that exposes two procedures, `ADD` and `SUB`, with procedure numbers 1 and 2. The triple of program, version,

and procedure number uniquely identifies a callable procedure. It appears in the header of every call and is what Lauberhorn uses to demultiplex an incoming request to the correct registered handler.

```
program MATH_PROG {
    version MATH_VERS {
        add_resp ADD(add_call) = 1;
        sub_resp SUB(sub_call) = 2;
    } = 1;
} = 0x12345;
```

Listing 2: ONC RPC/XDR Representation of a program with two procedures.

Definitions such as Listings 1 and 2 are conventionally placed in a `.x` file and fed to `rpcgen`, which generates the corresponding C data types along with serialization routines and client/server stubs. The runtime reuses the generated XDR serializers in its codec interface (Section 5.3).

Call and Reply Messages

Every RPC call consists of a header followed by the XDR-encoded message. In addition to the identifying triple, the call header (Figure 2) contains:

- **xid**: a transaction identifier, freely chosen by the client and placed at the beginning of the message. The client uses it to match an incoming reply to the request that generated it.
- **RPC version**: always 2, as this is the version Lauberhorn supports.
- **Authentication info**: since Lauberhorn only handles unauthenticated messages, the final 16 bytes of the header are zero.

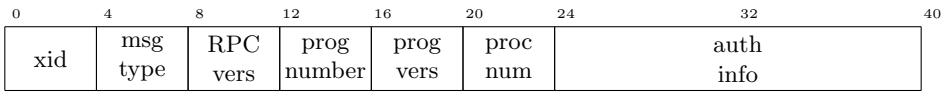

Figure 2: Structure of the 40-byte ONC RPC call header without authentication (offsets in bytes).

A reply likewise consists of a header followed by the XDR-encoded result message. The reply header echoes the call's xid, allowing the client to match it to the outstanding request, followed by status fields indicating whether the call was accepted and executed successfully.

3.3 Dandelion

Part of this thesis rewrites components of Dandelion to invoke functions via Lauberhorn. This section gives a brief overview of Dandelion's programming model and architecture. Section 6 then describes the changes made to them. A complete overview of Dandelion is provided by Kuchler et al. [6].

Dandelion is a serverless function platform designed for elasticity. Sandboxes exposing a POSIX-like interface are expensive to start. To keep request latency low, they are therefore often pre-provisioned, which in turn increases idle resource consumption. Dandelion avoids this tradeoff by only exposing a restricted, cloud-native programming model instead. It accepts only *pure compute functions*, i.e. user code that takes in a set of inputs and produces a set of outputs without invoking any system calls. All interaction with the outside world (currently limited to HTTP) happens through platform-provided *communication functions*, which are trusted and run outside of sandboxes. Because pure compute functions do not require a guest OS, Dandelion can execute them in lightweight sandboxes (e.g. processes or KVM virtual machines) that cold-start in hundreds of microseconds [6].

Programming Model

Users express applications as *compositions*. Compositions are directed acyclic graphs whose nodes are compute functions, communication functions, or other compositions. Each function takes and returns a variable number of arguments, where each argument is a *set* of items. An edge of the graph connects an output set of one function to an input set of another, together with a keyword describing how the set is distributed to consumers. Among others, `all` passes the entire set to a single instance, while `each` splits the set into *shards* - one item per instance - causing the consumer to be instantiated once per item, in parallel.

Figure 3 illustrates this with a matrix-matrix multiplication expressed as a composition of two functions.

The composition takes an argument `mat_in`, a set containing a single $N \times N$ matrix, and an argument `vecs`, a set of N vectors of length N. The `all` keyword passes the full matrix to every instance of `mat_vec_mul`, while `each` shards `vecs`, so that `mat_vec_mul` executes N times, once per (matrix, vector) pair. The resulting N products are collected into one set and passed with `all` to a single instance of `mat_reduction`, which combines them into the output matrix.

```
function mat_vec_mul(matrix, vector) => (product);
function mat_reduction(vectors) => (matrix);

composition mat_mat_mul (mat_in, vecs) => (mat_out) {
    mat_vec_mul(matrix = all mat_in, vector = each vecs) => (res_tmp = product);
    mat_reduction(vectors = all res_tmp) => (mat_out = matrix);
}
```

Listing 3: Matrix Multiplication Composition.

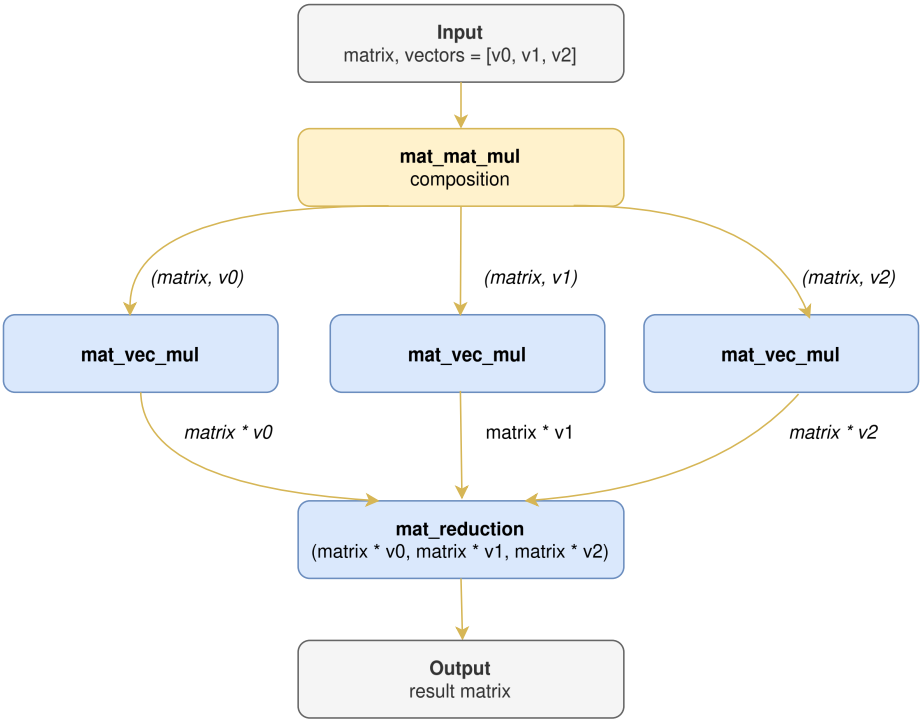

Figure 3: Matrix multiplication composition in Dandelion.

Architecture

A Dandelion worker node consists of a front-end, a dispatcher, and a set of engines [6]. All client interaction - registration of functions and compositions as well as their invocation - enters through the HTTP front-end, which forwards requests to the dispatcher and eventually returns results to the client. The dispatcher schedules functions on compute engines and maintains a registry of pre-registered functions and compositions. *Compute engines* execute functions in an isolated context (e.g., a process or KVM virtual machine), one at a time and to completion, and return the results to the dispatcher, which passes them on to dependent functions or back to the client via HTTP. All data passed to and from the HTTP front-end is serialized as `BSON` (Binary JSON).

4 Lauberhorn Kernel Module

Lauberhorn ships with a kernel module that configures the hardware and exposes it to the rest of the system. Like a regular NIC, Lauberhorn appears as a network device through the Linux `net_device` interface. Additionally, it exposes a character device, `/dev/lauberhorn`, which provides accelerator-specific controls such as registering and deregistering RPC endpoints (Section 3.1).

An application that wants its RPC handlers scheduled by Lauberhorn interacts with the module as follows:

1. The process opens `/dev/lauberhorn`, which registers it with Lauberhorn as an *RPC application*.
2. Each worker thread `mmap`s a region of NIC-homed memory through which it communicates with the NIC (Section 3.1).
3. The process registers RPC endpoints and their handlers via `ioctl` on the character device.

To test the runtime while the hardware was not yet ready, we extended this kernel module to mock Lauberhorn's behavior in software. The user-facing API, the character device and the `ioctl` interface are kept identical. A functionally correct software implementation was a prerequisite for the rest of this work. The mock must replicate the following behavior:

- **Fast Path**: intercept incoming RPC packets destined for registered endpoints before they reach the kernel's network stack.
- **Bypass**: deliver all non-RPC packets unchanged through the host's network stack.
- **Scheduling**: deliver intercepted RPC calls to a worker thread of the correct application, scheduling that thread onto a worker core if necessary.
- **Reply Routing**: send RPC replies back to the originating client / deliver RPC replies from nested calls to the correct process.

Figure 4 shows how these components interact on an incoming packet in the software implementation. The following subsections describe each in turn.

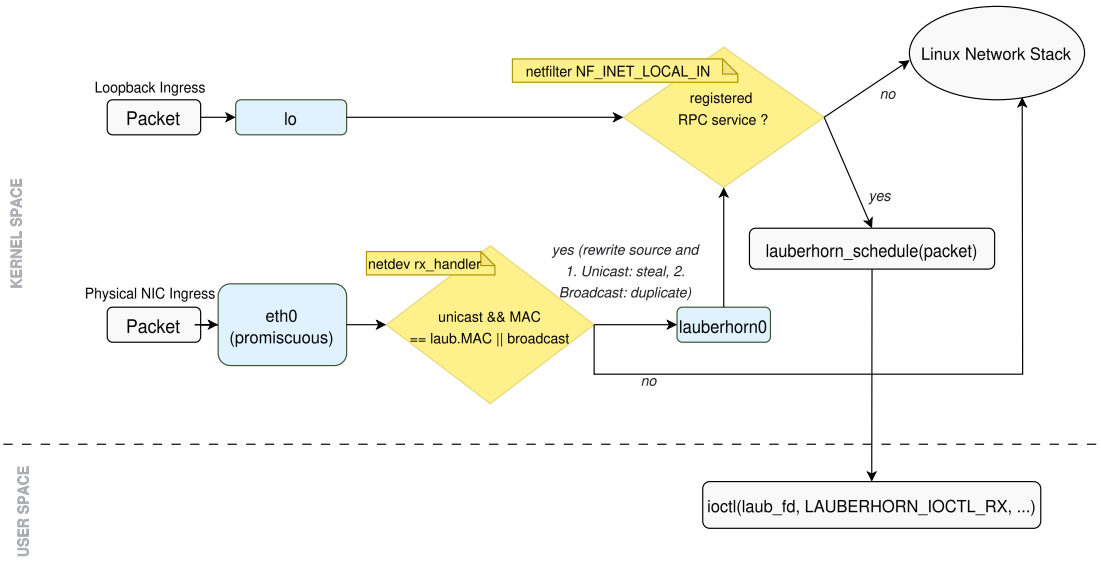

Figure 4: Flow of incoming packet through the kernel module

4.1 Network Device

The real kernel module already exposes Lauberhorn as a network device (Section 3.1). For the mock, where no hardware backs it, the `lauberhorn0` device is instead bound to a physical interface (e.g., `eth0`) through which the actual traffic flows. A net-device `rx_handler` on this interface inspects all incoming frames. Unicast frames addressed to the Lauberhorn MAC are reassigned to `lauberhorn0`, while broadcast frames such as ARP requests are cloned and the copy is injected into the receive path of `lauberhorn0`. Together with strict ARP settings (`arp_ignore=1`, `arp_announce=2`), each interface answers ARP only for its own address with its own MAC, so that peers resolve addresses in the Lauberhorn subnet correctly. All other traffic passes on to the kernel network stack unchanged.

This mirrors the behavior of a `macvlan` interface but is built directly into the Lauberhorn kernel module. Since the real module already implements the `lauberhorn0` network device, binding it to a lower device lets the mock extend that existing code rather than introducing a separate virtual device alongside it.

4.2 Fast Path

In hardware, RPC packets matching a registered endpoint never traverse the kernel's networking stack. The NIC parses each request and dispatches it to a

worker core by answering that core's stalled load on a NIC-homed cache line (Section 3.1). We implement this via a netfilter hook at the NF_INET_LOCAL_IN stage that intercepts packets reassigned to lauberhorn0, as well as loopback packets destined for the Lauberhorn subnet. Intercepting loopback traffic allows client and server to run on the same machine. For each selected packet, the module checks whether it is a valid RPC message for a registered handler. If this is the case, we withdraw the packet from the network stack and invoke the software scheduler (Algorithm 2) which either delivers the packet to a core that runs a thread eligible to handle the call or enqueues it in a per-process packet queue.

User-space workers obtain and answer these packets through two `ioctl` commands that replicate the ECI cache line exchange: `LAUBERHORN_IOCTL_RX`, with which a thread requests its next packet (the analog of the stalled load on the NIC-homed cache line), and `LAUBERHORN_IOCTL_TX`, with which it submits a reply (the analog of writing one).

4.3 Bypass

Packets that reach the netfilter hook but fail the RPC checks (no registered endpoint, or not an RPC message at all) are left untouched and continue through the network stack as normal. The Lauberhorn device, therefore, remains usable as a conventional network interface. On real hardware, bypass traffic is steered to a dedicated core that injects it into the kernel's network stack. We deliberately mock this less faithfully than the fast path as its exact mechanism does not affect the properties we evaluate.

4.4 Reply Routing

Lauberhorn stores connection state in hardware tables. For every incoming call, the NIC records the (xid, source IP, source Port) tuple which it looks back up when it sends out a reply for a matching xid. For outgoing calls, the NIC records the (xid, source PID, source Port, dest. Port, dest. IP) tuple which it uses to route incoming replies back to the correct application / process. To avoid maintaining state that mirrors the hardware in the kernel module, we attach all information needed to route a message directly to the message itself.

Replies to incoming Calls

When a call is delivered to user-space, the kernel module attaches the connection information parsed from the packet headers (source IP, source Port) to the packet descriptor. The runtime echoes this information back unchanged in the descriptor of the corresponding reply, and the kernel module builds the reply's headers directly from it. The kernel module therefore never tracks which calls are in flight.

Replies to nested Calls

When the reply to a nested call arrives, the kernel module must determine whether the packet is a nested-call reply at all (it could be a reply to a client

running on the same machine), and which process issued the original call.

The first is decided by port. All nested calls of all processes are sent from a single port, reserved at module load time. Any RPC reply arriving on this port is by construction a nested-call reply.

The second is determined via `xid`. On transmission, the kernel module encodes the index of the sending process in its process table into the top four bits of the `xid`. Before delivering the packet to user-space, the top 4 bits are zeroed. From the runtime's perspective, the top four bits are required to always be 0 when choosing an `xid` for outgoing calls.

It was once discussed whether hardware should equally rely on this (i.e., one fixed port, PID encoded into the top bits of the `xid`) to avoid allocating new per-call state on every outgoing call. However, such a design seems protocol-invasive and would require clients to cooperate with a fixed convention. It would further tie the maximum number of active services to the number of reserved bits in the `xid`. This idea was discarded in favor of a general connection table as described above.

4.5 Scheduler

Scheduling decisions are taken at exactly two points: when a worker requests its next packet via the `LAUBERHORN_IOCTL_RX ioctl` on the character device, and when a packet arrives. The RX call is the only point at which a worker thread can be suspended, mirroring the hardware, where a core can be suspended only after issuing a load on the NIC-homed cache line.

When a thread issues an RX request, the module first checks whether any registered application process has a non-empty packet queue, and if so, selects the process with the longest queue. If that is the caller's own process, the packet at the head of its queue is returned immediately. Otherwise, the caller marks a thread of the selected process as runnable and yields execution (Algorithm 1). This policy always serves the largest backlog first, which can starve a service whose queue stays persistently shorter than another's. In our experiments, this does not arise, as the testing workload distributes requests equally across services, ensuring queues remain approximately balanced. Since our goal is to test application correctness rather than to evaluate scheduling policy, this was considered sufficient.

Algorithm 1: Worker RX path via `LAUBERHORN_IOCTL_RX ioctl`.

Input: calling thread t of application P
Output: next packet for t

```
1  B ← longest_queue() ;            // application with the longest packet queue
2  if B = P then
3      return dequeue(P.pkt_queue)
4  else if B = NULL then
5      simulate wait on cache line by spin-waiting on work for P or until
         time-out;
6      return dequeue(P.pkt_queue) ;                    // NULL on timeout
7  else
       // a busier service exists: yield this core to it
8      t' ← find_free_thread(B);
9      deschedule(t); schedule(t', this core);
       // t resumed here when re-scheduled, only happens if scheduler in (Alg. 2)
          pushes to P's queue and wakes t up
10     return dequeue(P.pkt_queue)
```

The complementary path is triggered by packet arrival, shown in Algorithm 2. The scheduler scans the worker cores and places the packet on the first suitable one, in order of preference: an idle core, then a core whose waiting thread already belongs to the packet's application, and finally, if neither exists, a core waiting on a thread of a different application, which is preempted in favor of the target.

Algorithm 2: Packet placement on arrival.

Input: arriving packet *pkt* for application/process P

```
1  victim ← NULL;
2  foreach worker core c do
3      if c has no resident thread then
           // idle core: schedule a free thread of P and deliver
4          t ← find_free_thread(P);
5          if t ≠ NULL then enqueue(P.pkt_queue, pkt); schedule(t, c);
            return;
6      else if c's thread is waiting and belongs to P then
           // same application: deliver directly. Thread will receive this as it
              spin-waits on P's queue (Alg. 1, L5)
7          enqueue(P.pkt_queue, pkt); return;
8      else if c's thread t' is waiting and victim = NULL then
           // different application: remember as fallback victim
9          victim ← c;
10 if victim ≠ NULL and t ← find_free_thread(P) ≠ NULL then
       // preempt the thread running on the victim core in favor of t
11     enqueue(P.pkt_queue, pkt);
12     deschedule(t'); schedule(t, victim);
13 else
       // no core available: queue the packet for P
14     enqueue(P.pkt_queue, pkt);
```

4.6 Module Setup

The module ships with a utility script, `setup.sh`, which inserts the kernel module and configures its parameters: the physical interface to which Lauberhorn binds, the MAC and IP address of the Lauberhorn device as well as the port that is used for outgoing RPC messages.

```
setup.sh --lower-dev cpu40g1 --ipaddr 192.168.129.222 --mac 00:00:00:00:00:02
↪  --port 11111
```

Listing 4: Example invocation of `setup.sh`.

The script additionally applies the ARP-related `sysctl` settings that the address-resolution scheme of Section 4.1 relies on.

5 Lauberhorn Runtime

As mentioned, the Lauberhorn software platform ships with a user-space runtime library exposing an API for interacting with the NIC to do tasks such as registering and deregistering RPC endpoints or starting worker threads scheduled by the NIC. While the public API changed little during this thesis, significant parts of the implementation were rewritten and extended. The most notable additions are support for nested RPC calls and a more general serialization interface. A sample program using the described API is provided in the Appendix.

The following sections describe the runtime's essential features in the order an application is likely to use them: registering with the NIC as an RPC server application, registering services, starting worker threads, and issuing nested RPC calls. For each, we introduce the API, explain the implementation, and highlight any design decisions made.

5.1 Registering with the NIC

On startup, each process calls `lauberhorn_init` once. It opens `/dev/lauberhorn` which registers the process with Lauberhorn as an RPC server *application* (Section 4) and initializes the runtime's global state.

5.2 Service Registration & Deregistration

Before Lauberhorn can schedule and invoke any handlers on a worker core, the application must register at least one service via `lauberhorn_reg_srv` (Listing 5), which returns an integer service identifier for later deregistration with `lauberhorn_dereg_srv`.

```
typedef enum { EXEC_FIBER, EXEC_INLINE } lauberhorn_exec_ctx_t;

typedef struct {
    void *data;
    void* (*func)(void* priv, void* msg, int xid);
    void (*free)(void* result);
    lauberhorn_exec_ctx_t mode;
} lauberhorn_handler_t;

int lauberhorn_reg_srv(
    lauberhorn_t *ctx, lauberhorn_handler_t handler,
    uint32_t prog_num, uint32_t prog_ver,
    uint32_t proc_num, uint16_t listen_port,
    rpc_codec_t *rpc_codec
);

int lauberhorn_dereg_srv(lauberhorn_t *ctx, int srv_id);
```

Listing 5: Lauberhorn service registration API.

To register a service, the following information is needed:

1. **Endpoint Information**

   An RPC endpoint follows the ONC RPC conventions (Section 3.2). It is identified by the (program number, program version, procedure number) triple.

2. **Handler**

   The server registering the function must of course also provide the handler which should be invoked (`handler.func`). The handlers generated by `rpcgen` (Section 3.2) are functions which return a pointer to the result. To avoid memory leaks, this is done by statically allocating the response structure. However, as will become evident in Section 5.5, handler functions which process nested RPC calls need to be fully re-entrant, which means that returning a pointer to a statically allocated result buffer is not generally possible anymore.

   The application therefore also registers `handler.free`, which the runtime invokes once the result is no longer needed. For handlers returning static storage, a no-op suffices. `handler.data` is an opaque pointer passed to the handler on every invocation. Finally, `handler.mode` selects the execution context. `EXEC_INLINE` runs the handler directly in the worker thread, while `EXEC_FIBER` runs it in a suspendable fiber, which is required if the handler issues nested calls. This is discussed in more detail in Section 5.5.

3. **Serialization**

   The `rpc_codec` argument supplies the routines Lauberhorn uses to serialize and deserialize messages as hardware serialization / deserialization is still pending. This is discussed in more detail in Section 5.3.

State registered with the NIC

Lauberhorn dispatches a request by handing the worker a pointer associated with the endpoint (Section 3.1). Since the runtime needs more than the bare function pointer, such as the serializing routines or the private data, the pointer passed to Lauberhorn refers to a heap-allocated descriptor holding the handler together with the further required metadata (Listing 6).

```
struct lauberhorn_hw_handler {
        // handler
        lauberhorn_handler_t handler;
        // holds info about the service (serializers, etc.)
        struct srv* srv;
};
```

Listing 6: Structure registered with the NIC.

Service Deregistration

To deregister a service, we can simply call `lauberhorn_dereg_srv` with the integer that `lauberhorn_reg_srv` returned. Deregistration currently has no safe semantics. The hardware would have to flush all pending packets for the service, and the allocated handler descriptor (Listing 6) must not be freed while invocations of the service are still running - either by waiting for them to complete or by forcibly terminating them. Deregistering a service with in-flight or pending requests is therefore undefined behavior at present. The same currently applies more broadly. The kernel module's registration and deregistration code is not thread-safe, so registration and deregistration calls must be externally serialized.

5.3 Serialization & Deserialization

As mentioned, the user must supply serializers and deserializers for each registered service. The initial version of the runtime required the user to register a serializer and deserializer of type `bool(*xdrproc_t)(XDR*, ...)` as is generated by `rpcgen` (Section 3.2). When deserializing data, the `rpcgen`-generated XDR routines automatically allocate memory for any nested structures such as arrays.

`<rpc/xdr.h>` exposes `xdr_free` which frees any memory that may have been dynamically allocated during deserialization. However, `xdr_free` must be called only after the handler has returned, as we can only then guarantee that the data is not used anymore. Therefore, it is the runtime's responsibility to invoke this function.

This introduced problems when registering functions with Lauberhorn where the generated serialization and deserialization routines conform to the XDR standard, but are implemented in another language, such as Rust – the language in which Dandelion is written. In such a scenario, the memory dynamically allocated by the deserializer is managed by the Rust runtime, while `xdr_free` that is invoked by the Lauberhorn runtime and exposed by `libtirpc` expects memory allocated by libc's `malloc` (or similar).

This would effectively restrict a user to being only able to register deserializers whose allocated memory can be freed with `xdr_free` from `libtirpc`. This would make development in another language more complex and dependent on FFI layers.

To avoid this, rather than registering only serializers and deserializers, the server must now additionally register a `free` function that the runtime invokes on the deserialized data once it is no longer needed. To further reduce the coupling to XDR, it was decided to make the serialization interface more general, as shown in Listing 7.

The server can now supply custom methods for serializing and deserializing both calls and replies as well as for allocating memory to hold the deserialized arguments. The need for reply deserializers and call serializers arises from the introduction of nested RPC calls, as described further in Section 5.5.

```c
typedef enum { RPC_FREE_CALL, RPC_FREE_RESP } rpc_free_kind_t;
typedef enum { RPC_ALLOC_CALL, RPC_ALLOC_RESP } rpc_alloc_kind_t;

typedef struct {
    // serializers for call / reply
    int (*marshal_call)(void *in_buf, void *out_buf, size_t in_len, void* priv);
    int (*marshal_resp)(void *in_buf, void *out_buf, size_t in_len, void* priv);

    // deserializers for call / reply
    int (*unmarshal_call)(void *in_buf, void *out_buf, size_t in_len, void
    ↪  *priv);
    int (*unmarshal_resp)(void *in_buf, void *out_buf, size_t in_len, void
    ↪  *priv);

    // free unmarshaled data / alloc buffer to unmarshal into
    void (*free)(void *data, rpc_free_kind_t kind, void *priv);
    void *(*alloc)(rpc_alloc_kind_t kind, void *priv);
} rpc_ops_t;

typedef struct {
    rpc_ops_t *ops;
    void *priv;
} rpc_codec_t;
```

Listing 7: Lauberhorn Serializing Interface.

Serialization

The serializing procedures receive a pointer to a pre-allocated, runtime-owned buffer and serialize into it. In Lauberhorn, this buffer is currently 1500 bytes long, as messages are UDP based and bounded by the standard Ethernet MTU (Section 3.2). Should longer messages need to be supported in the future, the maximum buffer size could be made configurable via a command-line parameter or via a dedicated entry in a configuration file.

Deserialization

Deserialized data can be arbitrarily large, so the runtime first obtains a destination buffer via `rpc_ops_t.alloc` with the appropriate kind (`RPC_ALLOC_CALL` or `RPC_ALLOC_RESP`), then invokes the unmarshal routine to decode into it. When the data is no longer needed, the runtime releases it via `rpc_ops_t.free` with the corresponding kind.

Private Data

In addition to the serialization and deserialization procedures, the server may register a pointer to opaque private data in `rpc_codec_t` which is passed to every method on invocation. This makes it straightforward to wrap existing serializers, such as those generated by `rpcgen`, whose signatures differ from those required by `rpc_ops_t`. As all testing code for the runtime relies on XDR-based serializers generated via `rpcgen`, Lauberhorn provides a dedicated

adapter for `rpcgen`-generated serializers, which defines the required procedures and wraps them into an `rpc_codec_t` instance.

Workflow

Given the registration of a function with an instance of `rpc_codec_t`, the marshaling and unmarshaling flow for an incoming call proceeds as illustrated in Figure 5.

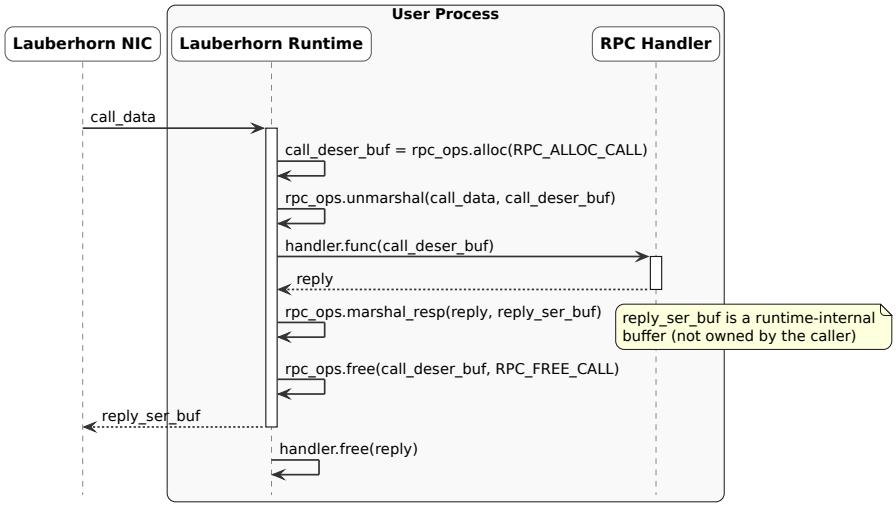

Figure 5: Lauberhorn Call - Marshaling & Unmarshaling (simplified API).

Hardware Serialization

As mentioned, Lauberhorn will eventually come with serialization and deserialization support in hardware. Once this is available, the `rpc_codec_t` instance will not be required anymore, as all of its functions will effectively decay to no-ops. To enable use-cases where hardware serialization is explicitly not desired, for example because the services require serialization/deserialization protocols that are not supported by hardware, `rpc_codec_t` may also be kept, where a new parameter indicates if hardware serialization should be enabled or not – and if not, the methods from the provided `rpc_codec_t` instance would be used by the runtime as is currently the case.

5.4 Starting Workers

Once the server has registered endpoints as described in Section 5.2, worker threads can be started. Each invocation of `lauberhorn_create_worker` will

spin up a new thread which executes a loop where it waits for data to come in from the NIC, executes the handler and sends the results back out.

Since Lauberhorn only has control over a fixed number of isolated worker cores (currently 4 by default), any process can at most start 4 worker threads, as any more would not be able to run concurrently anyway.

For each worker, the user additionally has the option to register callbacks which are invoked immediately after startup and right before exit of the worker (i.e. when the runtime shuts down).

```
typedef void (*lauberhorn_user_cb_t)(int);

lauberhorn_worker_t lauberhorn_create_worker(
        lauberhorn_t* ctx, lauberhorn_user_cb_t init, lauberhorn_user_cb_t fini
);
```

Listing 8: Create Lauberhorn Worker API.

5.5 Nested RPC Calls

Lauberhorn now supports nested RPC calls, where a service acting as an RPC callee can simultaneously initiate its own outgoing requests. The support for nested RPC calls was responsible for the biggest code changes and additions to the runtime library. Therefore, this section will be a bit more detailed and will go into more implementation details.

API

Programs issue nested calls through an asynchronous interface, in one of two styles.

1. Future-based style (Listing 9):

   `lauberhorn_call_async` dispatches a call to an endpoint described by an instance of `lauberhorn_rpc_endpoint_t` and binds a caller-provided `lauberhorn_future_t`. A future can be awaited individually with `lauberhorn_future_await`, or collected together with others into a `lauberhorn_future_set_t` and awaited as a group. `future_set_await_any` then blocks execution until any future in the set completes, removes it from the set, and returns it. The completed future holds an `rpc_result_t` with the deserialized reply data and a status code.

   Calls currently time out after one second, completing the future with `NULL` data and status `LAUBERHORN_RPC_TIMEOUT`. The timeout value can easily be made configurable if required.

   Ownership of the returned data passes to the caller, who releases it with `lauberhorn_free_result`.

2. Callback-based style (Listing 10):

   `lauberhorn_call_cb` registers a handler that the runtime invokes with the deserialized reply when it arrives, so the caller need not await at all. This is useful for fire-and-forget patterns where a reply may only be logged or should invoke follow-up work. Like service handlers, the callback declares its execution mode (`EXEC_INLINE` or `EXEC_FIBER`). Ownership of the deserialized reply remains with the runtime which automatically frees it after the callback returns.

```
// endpoint information
typedef struct {
    const char *daddr;
    uint16_t dport;
    uint32_t prog_num;
    uint32_t prog_ver;
    uint32_t proc_num;
    rpc_codec_t *rpc_codec;
} lauberhorn_rpc_endpoint_t;

typedef struct {
    void *data;                    // unmarshaled reply data
    rpc_codec_t *codec;            // needed by lauberhorn_free_result
    lauberhorn_rpc_status_t code; // SUCCESS, TIMEOUT, ...
} rpc_result_t;

int lauberhorn_call_async(lauberhorn_rpc_endpoint_t *ep,
                          void *payload, size_t payload_len,
                          lauberhorn_future_t *f);

lauberhorn_future_t *future_set_await_any(lauberhorn_future_set_t *s);
void lauberhorn_free_result(rpc_result_t *r);
```

Listing 9: Lauberhorn await-based async call API.

```
typedef struct {
        void(*cb)(rpc_result_t *reply, void*data);
        void *data;
        lauberhorn_exec_ctx_t mode;
} lauberhorn_cb_handler_t;

int lauberhorn_call_cb(lauberhorn_rpc_endpoint_t *ep,
    void *payload,
    size_t payload_len, lauberhorn_cb_handler_t handler
);
```

Listing 10: Lauberhorn callback async call API.

The Need for suspendable Handlers

Nested calls that are awaited require that the execution context issuing them can be suspended. Consider the worst case: every worker core is executing a handler, and every handler issues an outgoing call that targets a service running on the same machine. If handlers do not suspend execution while waiting for their replies, all cores are blocked and no core is free to process the incoming replies, as handlers run to completion and scheduling decisions are only made when a handler requests a new work item (Section 4). Thus, the system deadlocks.

Suspendable execution contexts are often obtained either by compiling handlers into a state machine where state that needs to survive across suspensions is heap-allocated, as done for example in Rust or JavaScript (stackless), or with user-space threads, as in Go's goroutines (stackful). Without compiler support, writing handlers in a hand-written continuation-passing style is difficult, so we chose user-space threads. The following paragraphs describe their implementation, their scheduling, and their integration into the worker loop.

User-Space Thread Model

A user-space thread (henceforth called fiber), like an OS thread, forms an independent execution context with its own stack but is completely transparent to the OS. A context switch only happens if a fiber yields cooperatively. This should happen at any point where execution would block while waiting for an external event. In Lauberhorn, this is exactly at the point when we await the reply from a nested call that has been dispatched.

Context switching between fibers is implemented using `ucontext_t` from `<ucontext.h>`, which provides the `getcontext`, `makecontext` and `swapcontext` primitives to save and restore execution states. The `ucontext` functions were removed from the POSIX standard in 2008 [9]. However, they remain available in glibc and were a perfect fit in terms of needed functionality for a first implementation of user-space threads.

One drawback is that saving a context with `ucontext` carries significant overhead. It saves not only the callee-saved registers, but also the floating-point state and the signal mask, the latter requiring a system call. A context switch via `swapcontext` can therefore take more than 1000 cycles [1].

Should the context switching overhead of `swapcontext` prove to be a bottleneck in practice, the fiber scheduler's switching primitives could be easily replaced by custom assembly or with `setjmp`, `longjmp` and a manual stack pointer initialization without changing the exposed API.

If the cost of switching stackful contexts is itself the limiting factor, the runtime could instead also expose a continuation-based API. Evaluating this alternative is left for future work.

Fiber Scheduling

Fibers are cooperatively scheduled by a dedicated *scheduling fiber*, one per worker thread. A worker registers itself as the scheduling fiber by calling `fiber_sched_init`. Unlike the fibers it schedules, it runs on the worker thread's original stack and all fiber switches on that thread pass through it. Each

scheduling fiber owns a thread-local FIFO queue of runnable fibers. Calling `switch_to(fiber)` suspends the scheduling fiber and transfers control to the selected fiber via `swapcontext`. From that point, the fiber runs uninterrupted until it returns control to the scheduling fiber by yielding or completing execution.

Before a fiber yields, it must register itself in the runtime as the waiter for some event (e.g. an outstanding RPC reply) such that it can eventually be re-scheduled by the runtime after occurrence of the event. Notification is edge-triggered. The scheduler wakes a waiting fiber exactly once, at the moment the event fires. A fiber must therefore re-check if the event is already ready after registering as a waiter and before actually yielding to prevent lost wake-ups.

This is achieved using a lock-free protocol. Each fiber carries an atomic word packing its state together with a generation counter, which is incremented when the fiber exits (as fibers are recycled). All transitions are single CAS (Compare and Swap) operations on this word, and every CAS requires the generation to be unchanged. A wake-up for a fiber that has since exited and been recycled fails the comparison and is ignored.

The protocol works as follows:

1. *Prepare:* the fiber sets its state to PARKING and only *then* re-checks the event.
2. *Cancel or commit:* if the event has already occurred, the fiber reverts to RUNNING and continues. Otherwise it yields to the scheduler, which completes the park by moving PARKING → PARKED.
3. *Wake:* the completion path (executing on any worker) calls `fiber_wake`. A PARKED fiber is made RUNNING and re-enqueued. If the fiber is still PARKING (i.e. it decided to sleep but the scheduler has not yet parked it) the waker instead marks it NOTIFY. The scheduler observes this when completing the park and re-enqueues the fiber immediately instead of parking it.

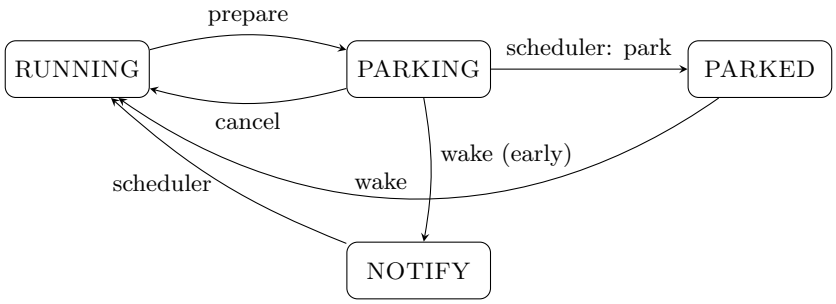

Figure 6: Fiber scheduling states.

Runtime Integration

At startup, the Lauberhorn runtime allocates a global table of reusable fibers to avoid allocating fresh stack memory on each request. Each fiber has an allocated stack of 64 KiB. The stack currently cannot grow dynamically and there is no mechanism in place to detect overflow.

Further, the runtime allocates a global *pending-call table* which holds one slot per in-flight nested call. Before a fiber dispatches an outgoing RPC call, it reserves a slot and registers its handle as the slot's waiter. The runtime later stores the reply in this slot and wakes the registered fiber, which extracts the result once it is re-scheduled.

Matching RPC Calls and Replies

As mentioned in Section 3.2, each ONC RPC reply consists only of the `xid` that was chosen at call time, some status information, and the message body. We therefore need a way to ensure that we can map an incoming RPC reply back to the fiber that originally called it. To avoid hash tables or similar map structures, we chose to encode the slot index that the calling fiber reserved in the table directly into the `xid`. At call time, the caller chooses the xid such that its lower 16 bits represent the index of the slot to which the fiber attached itself as a waiter, and the upper 16 bits are equal to the generation counter of the slot, which is increased by one every time a slot is reused.

On receiving a reply, the runtime looks up the slot indexed by the lower 16 bits of the xid and accepts the reply only if the upper 16 bits of the xid match the slot's current generation. This guards against stale replies. If a call times out and its slot is reused, a delayed reply to the original call still carries a matching slot index and would otherwise be delivered to the wrong caller. The generation check only fails if the slot is reused a multiple of 2^{16} times before the delayed reply arrives. This becomes exceedingly unlikely with sensible timeouts, though not impossible. [1]

Worker Loop

In the first version of the runtime, the worker loop simply waited on a cache line for a message, executed the handler function and returned the result. To support nested RPC calls, the worker loop had to be adjusted to integrate the fiber scheduler as shown in Algorithm 3.

When the loop starts, the ready queue is empty, so the worker immediately polls for a message from the NIC. Each received message is handled according to whether it is an RPC call or an RPC reply as described in more detail in Algorithm 4.

Periodically, the logic inside `handle_rpc_timeouts` sweeps the entire pending-call table and checks each entry for expiry. Timed-out entries are completed

[1] As explained in Section 4, the mock kernel module uses the top 4 bits of the `xid` to match replies back to the owning process. The per-slot generation counter therefore currently only occupies bits 16–27. This restriction disappears with real Lauberhorn hardware, which performs this matching differently.

through the same path as in `handle_packet`, with the reply set to `NULL` and the status to `LAUBERHORN_RPC_TIMEOUT`.

As long as the pending-call table remains small, this linear sweep carries low overhead. Should the table grow substantially, a more refined structure such as a timer wheel could replace the sweep without changing any externally visible behavior.

Algorithm 3: Worker main loop with fiber scheduling.

```
1 fiber_sched_init() ;                    // register this thread as its scheduling fiber
2 while Lauberhorn is running do
3     fiber_sched_run_ready() ;                          // run all runnable fibers
4     handle_rpc_timeouts() ;                            // check for call timeouts
5     if get_work(endpoint) then
6         handle_packet(endpoint) ;                      // dispatch call or reply (Alg. 4)
```

RPC Call

If the message is an incoming call, we could just initialize a new fiber with the call's handler function, push the fiber to the ready queue, and return. In the next iteration, the fiber queue will be non-empty and the handler will be executed immediately. However, this means that every incoming RPC call now pays the price of initializing a fiber and doing at least two context switches. Once to execute the fiber, and once to return.

Given that Lauberhorn was designed with performance and the least possible overhead in mind on the fast path, this is suboptimal. Since fibers were only introduced to make nested calling possible, non-nested functions do not necessarily need to be run inside a fiber but can be executed directly in the main fiber's execution context. Therefore, as already hinted at in Section 5.2, a user determines in which context a handler is executed by setting `EXEC_INLINE` or `EXEC_FIBER`.

This allows the dispatcher to take the fast path for all RPC calls which do not await any nested calls and performance should be (almost) identical to the original worker thread loop, apart from the few branches which check message type and the execution context.

RPC Reply

If the incoming message is an RPC reply, the reply is matched to a slot in the pending-call table based on the lower 16 bits of its `xid`, as described above. We then check if the reply should be handed back to the caller (call invoked via `lauberhorn_call_async`) or whether we should invoke an attached callback (call invoked via `lauberhorn_call_cb`). In the former, we notify the attached waiter of the packet arrival. In the latter, we execute the attached handler either inline or in a new fiber depending on the chosen mode.

Algorithm 4: Inbound packet dispatch (`handle_packet`).

Input: endpoint descriptor d delivered by the NIC

```
1  if d.type = RPC_CALL then
       // run in a fiber if nested, else inline on this thread
2      if d.mode = EXEC_FIBER then
           push(ready_queue, prepare_fiber(d.handler))
3      else execute_inline(d.handler, d.data)
4  else                                                    // RPC_REPLY
5      s ← slot_from_xid(d.xid)
6      if s = NULL or store_reply(s, d) fails then return   // unknown
           xid, or slot freed by timeout
7
8      switch s.kind do
9          case await do notify_fiber(s.waiter)
10         case callback do
11             if s.mode = EXEC_FIBER then
                   schedule_callback_fiber(s)
12             else run_callback_inline(s); release_slot(s)
```

Hardware-Managed Continuation State

We also considered offloading continuation state to the NIC. Currently, we encode a slot index into the `xid` and resolve it to a pending-call slot with an attached fiber handle in the runtime. Instead, the NIC could store an opaque pointer on an outgoing call (a fiber handle, or equivalently a continuation pointer together with its heap-allocated state) and return it with the matching reply, analogous to the metadata pointer it delivers on an incoming call (Section 3.1). The reply path would then dispatch straight to the referenced fiber, switching to it if idle or depositing the reply for it to collect otherwise. In this case, no pending-call table would be required to match incoming replies back to the calling fiber.

At the time, we did not pursue this further, especially since it was unclear how much state the NIC would allocate on each outgoing call. However, the design remains retrofittable and could be worth pursuing, given that the NIC now allocates per-call connection state on each outgoing call (Section 4) to match a reply back to its calling process. The runtime is agnostic to where a fiber originates, so the scheduling and worker-loop machinery described above is unchanged whether the fiber comes from the local table or is handed back by the NIC.

End-to-End Flow

With all components in place – the fiber scheduler, the pending-call table and the xid tagging to match replies to the original caller – a full end-to-end asynchronous RPC call (`lauberhorn_call_async`) is executed as shown in Figure 7.

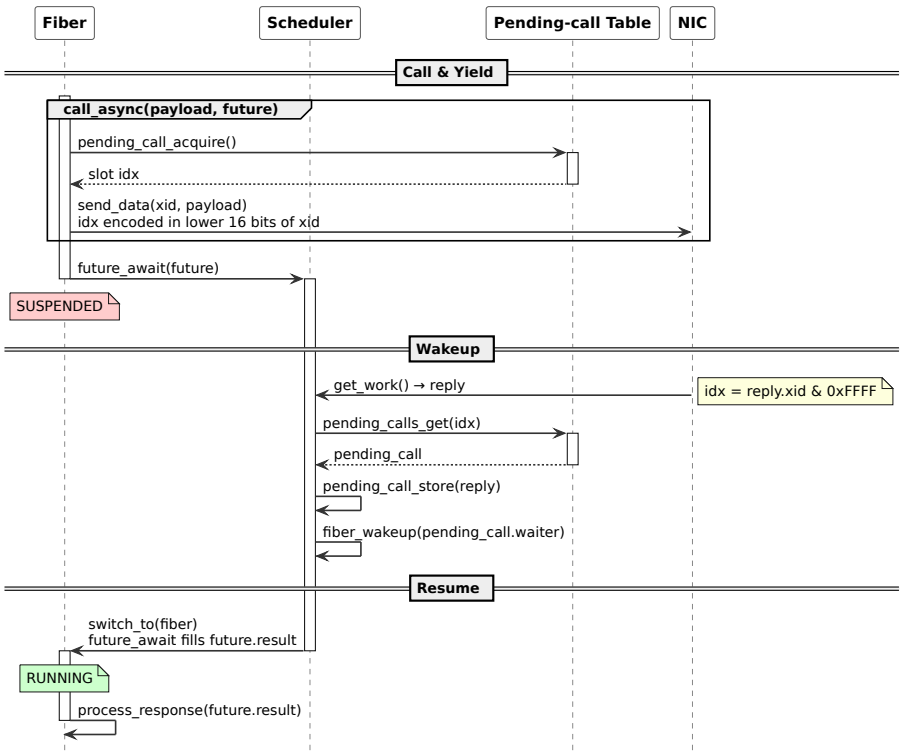

Figure 7: Lauberhorn asynchronous RPC flow.

6 Dandelion

To port Dandelion to Lauberhorn, several adjustments to the Dandelion runtime were necessary. The port does not cover all of Dandelion's functionality. We focused on what was needed to demonstrate Lauberhorn with a non-trivial application, namely executing single functions and compositions with nested calls. This section describes the main adjustments, the design decisions behind them, and the functionality that remains unsupported. The port is based on commit `98c4968` of the Dandelion repository (March 9, 2026) [5]. Since several interfaces changed upstream after this work began, the changes were not rebased onto `main`. Doing so would require adapting to the new interfaces, but no changes to the logic itself would have to be made.

6.1 Replacing the Dandelion Scheduler

The first design question was which role the Dandelion dispatcher should retain, i.e., how a packet should flow from arrival at the NIC to execution in one of Dandelion's isolation contexts. Two options were considered.

Option 1: Dispatcher as Entry Point

A natural first approach would be to keep the Dandelion dispatcher as a single point of entry for all incoming calls, and simply make it wait on a set of Lauberhorn cache lines. Lauberhorn would then be configured to deliver all incoming RPC calls to the dispatcher, from where they are scheduled onto compute engines as before. This would significantly reduce the porting effort, as the control plane and scheduling logic remain unchanged.

Although this approach could reduce latency for small packets, as we use Lauberhorn's cache-coherence protocol to deliver packets directly to Dandelion without going through the networking stack, it leaves Lauberhorn's core scheduling functionality entirely unused. The allocation of tasks to compute engines is still performed in software by the Dandelion scheduler, and Lauberhorn's ability to dispatch calls directly onto specific worker cores is not exploited.

Option 2: Full Hardware Scheduling

To exploit the full functionality of Lauberhorn, we decided to completely remove the Dandelion scheduler and expose handlers that directly execute a Dandelion function or composition on compute engines as RPC services.

Incoming calls are dispatched by Lauberhorn directly onto a worker core, with no software scheduling step in between. This changes Dandelion's structure substantially, since the dispatcher was its central component, but it is the only design in which Lauberhorn, rather than software, manages the scheduling decisions.

6.2 Architecture of the ported System

Figure 8 shows the remaining components of Dandelion and how they are connected to Lauberhorn.

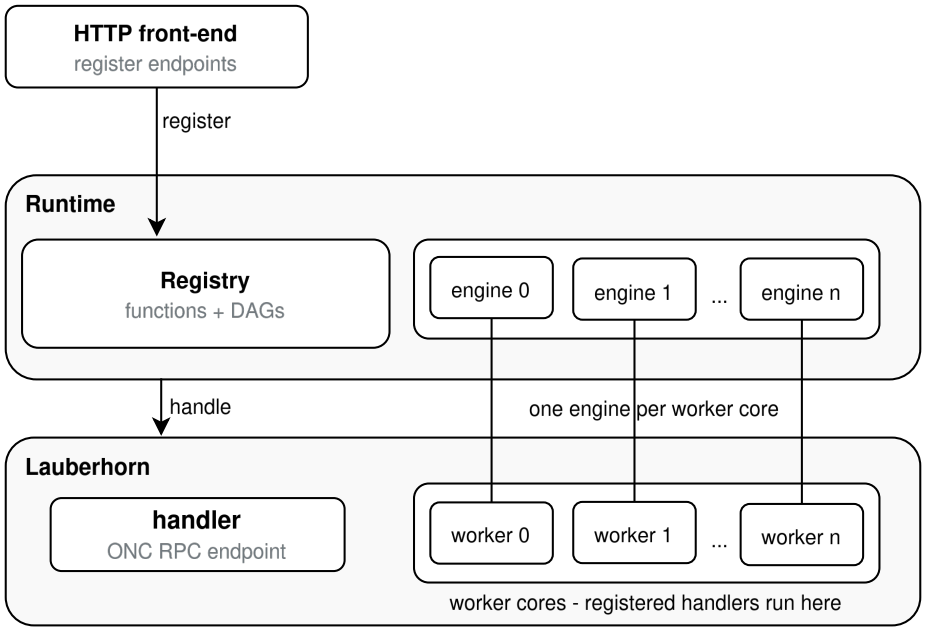

Figure 8: Architecture of Dandelion on Lauberhorn.

HTTP Front-end

The front-end remains virtually unchanged, but its role shrinks. Previously, all interaction with Dandelion, including every function invocation, passed through it. Since functions are now invoked and scheduled by Lauberhorn directly, the front-end retains only the two registration endpoints, `POST /register/function` and `POST /register/composition`. These could of course also be exposed as RPC endpoints registered with Lauberhorn.

Runtime

The runtime holds the system state shared by all components: a handle to the function registry, a vector of engines with one engine per worker core, and a handle to Lauberhorn for issuing commands.

Function and DAG registry

The registry stores the functions and compositions registered through the front-end. It remains in software as in upstream Dandelion and is owned by the runtime.

Engines

The engines remain unchanged. Each engine executes one function at a time in its isolation context (e.g., a process or a lightweight KVM instance). Because some isolation backends hold per-engine state, the runtime allocates one engine per worker core and each Lauberhorn worker uses the engine of the core it runs on.

Handler State

As described in Section 5.2, every function registered with Lauberhorn may carry private data. Dandelion functions need access to the registry and to the engines in order to set up an isolation context, so a reference to the runtime context is registered as the private argument of the Dandelion handler and passed to it on every invocation.

RPC Interface

Dandelion is exposed as an ONC RPC service with the request and reply types of Listing 11. A request names the function or composition to run and carries its serialized input. A reply carries the serialized output.

```
struct DandelionRPCRequest {
    string function_name<>;
    opaque data<N>;
};

struct DandelionRPCReply {
    opaque data<N>;
};
```

Listing 11: XDR Definition of the Dandelion RPC Interface.

6.3 Executing a Function

In a first version, every registered function and composition became its own RPC service. This immediately ran into a hardware limit. Lauberhorn supports only a small, fixed number of endpoint slots (currently 8). A handful of compositions of two or three functions each would exhaust the slots and severely restrict the workloads Dandelion could run.

The workflow of preparing and executing a function is identical for every function. We therefore register a single RPC handler at startup on a port and program triple set in Dandelion's configuration. Which function to run is carried in the request itself (Listing 11). The client includes the name of a previously registered function in `DandelionRPCRequest` alongside the input data.

When Lauberhorn delivers a request to a worker core, the handler deserializes it, looks the name up in the function registry, prepares the isolation context on that core's engine, loads the inputs, executes the function, and returns the serialized result (Figure 9).

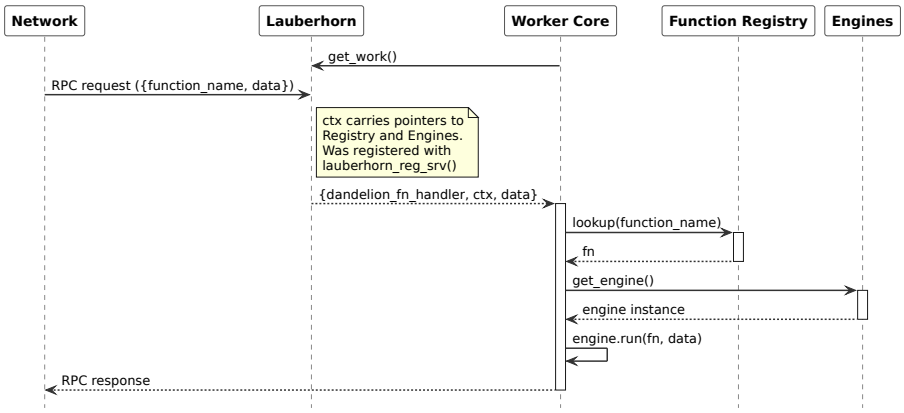

Figure 9: Invocation of a single Dandelion Function.

6.4 Executing a Composition

Compositions execute through the same entry point as functions. When the handler looks up the request's name and finds a composition rather than a single function, it does not execute on an engine directly. Instead, it acts as the composition's coordinator (Algorithm 5) which dispatches individual functions of the graph as separate asynchronous RPC calls.

As we only register a single endpoint which serves both compositions and single functions, the handler is registered with `EXEC_FIBER` (see Section 5.5) as compositions invoke and await asynchronous calls and must therefore be executed in a fiber context. As an optimization, we could register another endpoint internally which the dispatcher uses as an endpoint for function invocations to avoid execution in a fiber context if not needed.

Algorithm 5: Composition Coordinator Loop.

```
   Data: composition graph G, inputs
1  dispatch all shards of G's start nodes;
2  while any call is pending or any shard is still waiting to be dispatched
     do
3      wait for the next shard to complete;
4      if the node's last outstanding shard just completed then
5          combine its shards into the node's outputs;
6          dispatch any successor node whose inputs are now all available;
```

The coordinator is a data-flow scheduler over the composition graph. As described in Section 3.3, each input set to a function or a composition can be split into shards, each of which leads to an individual invocation of the same function.

We first dispatch calls for all shards of the composition graph's input nodes via asynchronous calls, and await any reply. Once all of a node's shard replies are back, they are reduced into its result set. Each output set is then provided as input to every dependent node, and once all of a dependent node's inputs are available, it is sharded and dispatched in the same manner.

For compositions with many nodes, some fanning out into many shards, every edge in the composition graph incurs a full serialization - deserialization round trip for both arguments and results as they are passed around via RPC calls and replies. Since all functions a composition invokes are currently registered on the same machine, this could be avoided. Intermediate results could be kept in a globally accessible table, with RPC payloads carrying only indices into that table. Data would then be serialized and copied only when entering or leaving a composition - rather than on every edge.

6.5 Unsupported Functionality

Beyond compute functions, Dandelion provides predefined communication functions (currently limited to HTTP, Section 3.3). We did not port them as they add nothing to demonstrating that a non-trivial application with single and nested function dispatch runs on Lauberhorn. Further, the runtime currently only supports awaiting RPC replies. Asynchronous HTTP could be added by monitoring completions either via a separate thread or inline in the event loop through an API such as `epoll`. Such requests would not benefit from Lauberhorn's fast path, but since the fiber wake-up mechanism is decoupled from the RPC layer (Section 5.5), extending it to further event sources should be straightforward.

7 Performance Evaluation

To evaluate the kernel module, the runtime, and the Dandelion port, we run two sets of experiments. First, we measure how request latency evolves as throughput increases. Second, we compare the end-to-end latency of different Dandelion functions and compositions against the upstream version.

7.1 Methodology

The server and client run on two separate Enzians, each booted with kernel 6.8.0 and the following configuration:

```
isolcpus=nohz,domain,managed_irq,44-47 nohz_full=44-47 rcu_nocbs=44-47
irqaffinity=0-43 kthread_cpus=0-43 rcu_nocb_poll
```

Cores 44–47 are fully isolated from the Linux scheduler, interrupt handling, and RCU callbacks and only run the worker threads spawned by the Lauberhorn runtime (Section 5.4).

To break down server-side latency, the kernel module records three per-request timestamps, appended to the reply:

1. handoff to the scheduler, right after the netfilter hook identifies the packet as an incoming RPC call or nested reply.
2. handoff to user-space, right before the `RX` ioctl returns.
3. return of the reply to the kernel, right after the `TX` ioctl.

We report (2)−(1) as *scheduling* latency which is the time from a packet being submitted to the scheduler to it being handed to user-space. (3)−(2) is reported as *user-space* latency, the time the request spends in the runtime. *End-to-end* latency is the round-trip time measured on the client.

7.2 Latency versus Throughput

To test the runtime and the kernel module, we measure latency as load increases with two configurations. In the **single-service baseline**, a single application running the adder service (A.1) receives all requests. No scheduling between competing applications occurs. This isolates the per-request dispatch and execution cost. In the **multi-service workload**, seven applications, all running the same adder service, are multiplexed across the four worker cores, with requests issued round-robin. Each arrival at a core is likely to target a different service than the resident one, forcing repeated process switches. Since per-request execution cost of the handler is identical, any difference between the two experiments isolates the overhead of scheduling and service switching.

The load is generated by a custom RPC client that asynchronously dispatches requests at fixed intervals to match a target rate. Each load level runs for 5 seconds, discarding the first 1,000 requests as warm-up, followed by a 10-second idle period. Every request in the reported runs received a reply and no packet drops were observed.

Single-Service Baseline

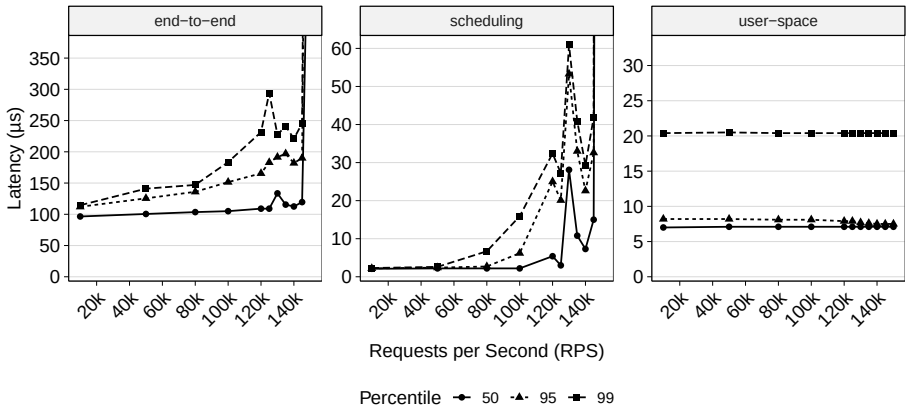

Figure 10: Latency versus load (kRPS) for the single-service baseline (across 4 worker cores). Note the differing y-axis scales.

Figure 10 shows latency as a function of requests per second (RPS). Table 1 details the low-load breakdown. The system sustains around 120,000–140,000 requests per second before latency spikes. The breakdown attributes the spike to scheduling latency. Once load exceeds the workers' service rate, packets queue in the kernel module and queuing delay grows without bound. The runtime's per-request latency remains flat across all load levels. The significantly higher P99 user-space latency comes from the periodic linear timeout-sweep across the pending-call table (Section 5.5) that was performed every 100 iterations.

Component	Single service (µs)			Multi-service (µs)		
	P50	P95	P99	P50	P95	P99
Scheduling	2.1	2.2	2.3	2.1	34.9	35.9
User-space	7.0	8.2	20.4	7.2	11.8	20.2
End-to-end	96.5	112.0	114.5	96.0	135.0	142.5

Table 1: Server-side and end-to-end latency at low load (50 kRPS single-service, 10 kRPS multi-service).

The runtime occupies a worker core for approximately 7 µs per request (P50 user-space latency, Table 1). Four worker cores should therefore sustain roughly $4 \times 10^6/7 \approx 570{,}000$ requests per second, assuming no other work on the critical

path. On the mock, the scheduler imposes a similar ceiling. Dispatch runs under a global lock, costing ∼2 µs per request (P50 scheduling latency, Table 1), which serializes scheduling to ∼500,000 decisions per second regardless of core count. In hardware the NIC makes this decision directly, so this cost becomes negligible.

The measured user-space latency covers only the time spent in the runtime. Before a core becomes schedulable again, the handler's reply must be wrapped into a packet, sent out through the kernel network stack, and control must be returned to user-space, where the worker calls back into the kernel to request the next packet. This adds to the critical path which is why the observed throughput is lower.

In hardware most of this overhead falls away. No syscalls, no packet assembly, no traversal of the OS network stack. The gap between initiating the reply and being ready to accept the next packet will likely shrink a lot, pushing throughput toward the ∼570,000 boundary.

Multi-Service Scheduling Overhead

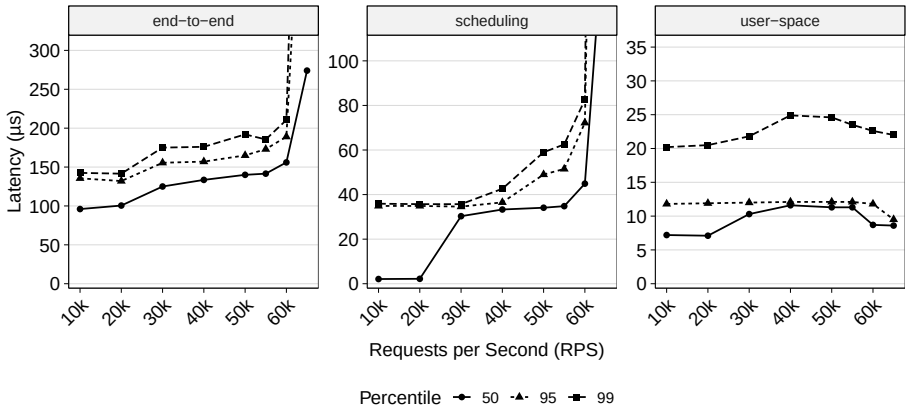

Figure 11: Latency versus load (kRPS) for the multi-service workload (across 4 worker cores). Note the differing y-axis scales.

Figure 11 shows the multi-service results. The workload saturates at approximately 60,000 requests per second. Each service switch, among others, adds a context switch to the critical path, which roughly halves sustainable throughput.

The median scheduling latency exhibits a distinct increase between 20 and 30 kRPS. This is a consequence of the chosen scheduling algorithm (Algorithm 2). When a packet arrives for a service with no thread spinning on a cache line waiting for work, the scheduler preempts the eligible core with the lowest ID. At low arrival rates, every core runs an idle thread by the time a new request arrives, so the lowest-numbered core absorbs all preemptions and cycles among

the services without a resident thread, while the other three cores keep their service. Most packets bypass preemption, and the median reflects scheduling without preemption, i.e. the same as in Figure 10.

Once the arrival rate is high enough, packets increasingly find the churn core busy and the scheduler falls through to the next core, evicting a previously resident service. Preemption spreads across all cores and the median scheduling latency jumps to $\sim$30 µs and stays flat until the dispatch path itself saturates.

7.3 Dandelion

To measure the performance of the Dandelion port, we measure end-to-end latency of different functions and compositions and compare them against the upstream version.

All functions are executed using the process-isolation backend and the same two-Enzian setup. Each workload runs 1,100 iterations of which the first 100 are discarded as warm-up. We evaluate three workloads: a single 2×2 matrix multiplication (`matmul`, depth 1) and two compositions of depth 2 and depth 4, where depth is the length of the critical path through the data-flow graph (A.3).

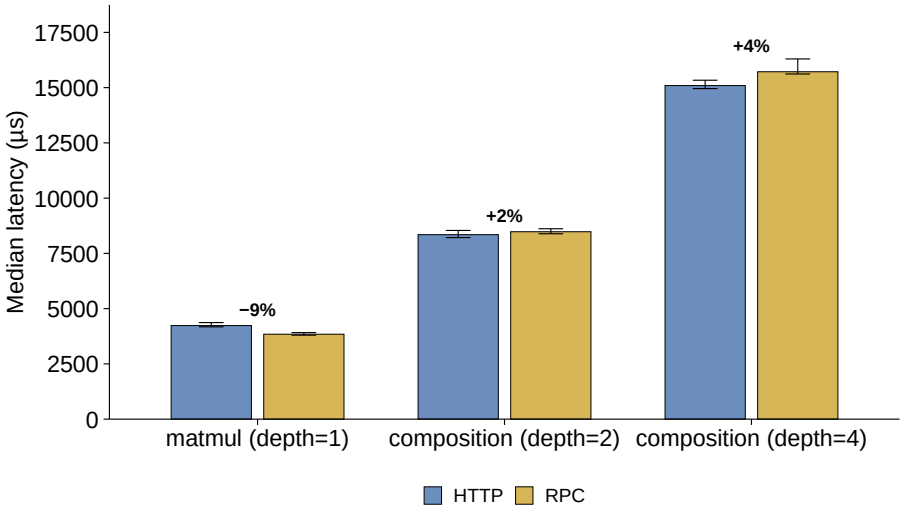

Figure 12: End-to-end latency of the port (RPC) versus upstream implementation (HTTP). Bars show median latency, error bars span the empirical 1st to 99th percentile.

Figure 12 shows the result. For the single matrix multiplication, the ported version is around 9% faster than the original implementation. These gains

likely come from a combination of bypassing the Dandelion scheduler, a shorter network path (the ported version withdraws the packet early at the netfilter hook (Section 4)), and lower protocol overhead than the baseline (UDP versus TCP plus HTTP).

For compositions, the port is around 2% slower at depth 2, and around 4% slower at depth 4. As described in Section 6, the port dispatches each node of a composition as a nested RPC call, which is routed back through the kernel via the loopback interface and pays a full argument serialization and deserialization. Since every node incurs roughly the same fixed serialization and deserialization cost for its arguments and results, the absolute added latency this causes should grow approximately linearly in the depth of the composition.

We did not instrument the port enough to separate the three contributions - scheduling, network round trip and execution - so the attribution is qualitative. On a hardware-backed Lauberhorn, where scheduling is offloaded to the NIC and the kernel is bypassed, the per-node overhead of a nested RPC call should shrink, and the remaining latency differences compared to the upstream version would mostly reflect execution overhead (e.g. the Dandelion scheduler). A precise breakdown, both in the mock and on hardware, is left for future work.

8 Scalability Limits

The runtime and the NIC each impose bounds on how far the current system scales, both in the number of services it can register and the number of requests it can have in flight.

As mentioned in Section 6, the number of services that can be registered simultaneously on the NIC is bounded by the size of its hardware table - a content-addressable lookup table, currently limited to 8 slots. Such tables cannot be scaled without bounds while keeping access latency low. Significantly expanding this table would likely require partitioning services into hot and cold sets. The NIC keeps metadata only for hot services in the fast lookup table, and offloads the remainder into ordinary address-indexed memory. Beyond the context-switching costs shown in Figure 11, a call to a cold service then additionally pays the cost of resolving the service to its memory address and fetching it.

The same limit applies to the tables tracking active connections (currently limited to 16 entries), for which the NIC allocates state on every incoming and outgoing call, as described in Section 4. Expanding these could similarly require offloading connection state into address-indexed memory, at the cost of a slower lookup.

Further limits arise in software. As discussed in Section 5.5, a fixed number of fibers is allocated at startup. Because contexts are stackful rather than stackless, a fiber must stay allocated for the entire lifetime of a call, including while it is suspended awaiting a nested call. A call that performs little work but issues a long chain of nested calls therefore occupies its fiber for far longer than it occupies a core. Under such workloads, the pool can be exhausted despite the worker cores not being fully utilized. A stackless implementation could partially alleviate this problem as the runtime would not need to keep stack memory around. However, the state that must be stored across suspensions still uses memory, albeit likely less than a full stack.

A way to reduce stack memory allocation would be to allocate a large virtual address range per fiber at initialization and let the OS lazily allocate pages as needed. Once a fiber is released back to the pool, allocated pages can be explicitly freed again.

A way to alleviate the fixed-pool limit would be to allow the fiber pool to grow dynamically up to a certain extent, to handle spikes in requests. However, the runtime still needs a policy for a certain upper bound. Requests arriving with no fiber available can either be dropped, which keeps latency bounded at the cost of completion, or queued until one is freed, which preserves the request but could lead to unbounded queuing delay. The same choice applies to the pending-call table. This table is currently also statically allocated with a fixed size and the runtime returns an error if we want to dispatch an RPC call and the table is full.

9 Related Work

The performance limitations of software-based RPC stacks have motivated numerous hardware-accelerated and kernel-bypass RPC systems (Section 2). We discuss two such systems, Dagger [8] and eRPC [4], focusing on their support for asynchronous nested calls, and briefly contrast how the NIC and the CPU exchange data in each design.

Dagger

Like Lauberhorn (once serialization in hardware is achieved), Dagger [8] offloads the entire RPC stack to an FPGA-based NIC that is attached to the host over a coherent memory interconnect instead of PCIe. The communication model between CPU and NIC, however, differs. Where Lauberhorn delivers a request by fulfilling a load on a NIC-homed cache line, Dagger allocates a pair of receive and transmit queues in host memory for every thread, and each thread continuously polls its receive queue. The NIC only observes these queues and has no knowledge of which threads are polling or executing, so it cannot place a request on a free core. It can only balance requests across the queues of a single service if that service runs multiple polling threads.

Like Lauberhorn, Dagger executes handlers directly on the thread that receives the request, and both blocking and non-blocking calls are supported. A handler can await one or several nested calls by polling for their replies. Blocking calls spin until the reply arrives, while replies to non-blocking calls are pushed into a per-client completion queue by a separate polling thread [7].

In any case, a handler that waits occupies its thread, and no new requests from the thread's receive queue are processed in the meantime. This limitation shapes the behavior of nested calls. Acyclic call chains cannot deadlock, because every service's dispatch threads are always runnable and guaranteed to make progress. However, a waiting handler blocks all requests queued behind it which means that cyclic call graphs can deadlock. In Lauberhorn, a handler that awaits yields its core back to the worker loop, which continues dispatching incoming calls and replies. Both acyclic and cyclic patterns are therefore handled, and waiting consumes no thread (Section 5.5).

Dagger's own evaluation illustrates the limits of non-suspendable handlers. A service that does little work itself but waits on calls to long-running services collapses in throughput, because the blocked handler prevents its thread from accepting new requests. Their remedy is to move request processing into separate worker threads scheduled by the OS, which restores throughput but adds inter-thread communication overhead.

Porting Dandelion compositions to Dagger could be achieved in two ways. First, a handler issues its nested calls and waits for the replies before returning - occupying the dispatch thread for the entire composition. Alternatively, handlers are rewritten to return early and be resumed once replies arrive, which requires turning each handler into a state machine (by hand or with Rust's `async` support) and extending Dagger's dispatch loop to resume them as completions come in.

eRPC

In contrast to Lauberhorn and Dagger, eRPC [4] is a general-purpose RPC library that requires no specialized hardware and runs on commodity NICs using kernel-bypass packet I/O. As in Dagger, each thread busy-polls its own send and receive queues from user space. The NIC has no notion of idle threads and cannot steer requests to free cores. Each application thread runs an event loop provided by eRPC, which polls for received packets, dispatches incoming requests to their registered handlers, invokes continuations for arrived replies, and transmits queued messages. Like Lauberhorn's worker loop, it drives both calls and replies on a single thread.

All calls in eRPC are non-blocking and continuation-based. A handler issues a call by enqueuing a callback that is invoked by the event loop once the reply arrives. For nested calls, a handler issues its requests and returns without a response. The response to the original request is later sent from a continuation. As in Lauberhorn, waiting handlers therefore do not occupy the thread, and cyclic call patterns cannot deadlock the system.

Porting Dandelion compositions to eRPC would likely be simpler than porting them to Dagger. The decomposition into continuations could again be generated by Rust's `async` machinery, but unlike in Dagger, no runtime extensions are needed to drive the resulting state machines. Each continuation could advance the state machine on invocation.

10 Conclusion

At the beginning of this work, the Lauberhorn hardware was not ready for testing yet, and the RPC runtime only supported non-nested calls for `rpcgen`-generated handlers. An initial implementation of the kernel module was necessary to start testing the runtime. The main challenge in designing the kernel module was how precisely the behavior of Lauberhorn should be mocked, as abstractions or simplifications could lead to behaviors that would not be observable with the real hardware (such as scheduling deadlocks). Implementing the scheduling of worker threads in the kernel module was particularly challenging, as it relies on having full control over the worker threads.

The runtime extensions were driven by what porting Dandelion required. The main addition was support for nested RPC calls, which lets a handler invoke nested functions and await them. To support this, we implemented a fiber scheduler that allows handlers awaiting a reply to be suspended. While other RPC runtimes such as eRPC use continuations to support this, we chose fibers as manually writing continuations without compiler support can be difficult.

A more general codec interface replaced the `rpcgen`-specific marshaling routines, so that XDR-conforming codecs can be written directly in other languages with memory allocation under the caller's control.

The runtime implementation still leaves some design decisions open. If we run out of pre-allocated slots for pending calls or fibers, any incoming request is currently dropped silently. It remains to be decided whether requests should instead be queued up to some bound or be accommodated by growing the pools on demand. Fiber stacks are also fixed in size, with no mechanism to detect stack overflow. Handlers could thus overrun the stack and corrupt runtime state. Ultimately, the idea of moving some continuation state into the hardware could be further considered, given that the NIC allocates connection state on each outgoing call.

Despite the remaining limitations, the kernel module, the extended runtime, and the Dandelion port form a working end-to-end system. The evaluation shows the runtime sustaining high throughput at per-request latency that stays flat under load. The Dandelion port can run arbitrary compositions on top of this, although it still glosses over some upstream features and does not handle errors gracefully on every path. What remains is replacing the mock with the hardware it stands in for, and a finer-grained instrumentation of the Dandelion port that can separate scheduling, network round trip, and execution time, to evaluate the benefit of having compositions scheduled by Lauberhorn rather than by Dandelion's own scheduler.

References

[1] Boost.org Context Switch Performance. https://www.boost.org/doc/libs/latest/libs/context/doc/html/context/performance.html. Accessed: 2026-07-11.

[2] Eisler, M. XDR: External Data Representation Standard. RFC 4506, May 2006.

[3] Haynes, T., and Noveck, D. Network File System (NFS) Version 4 Protocol. RFC 7530, Mar. 2015.

[4] Kalia, A., Kaminsky, M., and Andersen, D. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 1–16.

[5] Kuchler, T., Li, P., Zhang, Y., Cvetković, L., Goranov, B., Stocker, T., Thomm, L., Kalbermatter, S., Notter, T., Lattuada, A., and Klimovic, A. Dandelion. https://github.com/eth-easl/dandelion, 2025.

[6] Kuchler, T., Li, P., Zhang, Y., Cvetković, L., Goranov, B., Stocker, T., Thomm, L., Kalbermatter, S., Notter, T., Lattuada, A., and Klimovic, A. Unlocking True Elasticity for the Cloud-Native Era with Dandelion. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (New York, NY, USA, 2025), SOSP '25, Association for Computing Machinery, p. 944–961.

[7] Lazarev, N., Xiang, S., and Adit, N. Dagger. https://github.com/barabanshek/Dagger, 2021.

[8] Lazarev, N., Xiang, S., Adit, N., Zhang, Z., and Delimitrou, C. Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 36–51.

[9] Linux man-pages project. getcontext(3) — Linux manual page. https://man7.org/linux/man-pages/man3/getcontext.3.html, 2026. Linux man-pages 6.18, dated 2026-02-08. Accessed 2026-07-12.

[10] Pourhabibi, A., Sutherland, M., Daglis, A., and Falsafi, B. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2021), MICRO '21, Association for Computing Machinery, p. 407–420.

[11] Ramdas, A., Cock, D., Roscoe, T., and Alonso, G. The Enzian Coherent Interconnect (ECI): opening a coherence protocol to research and applications. *LATTE '21* (2021).

[12] Ruzhanskaia, A., Xu, P., Cock, D., and Roscoe, T. Rethinking programmed I/O for fast devices, cheap cores, and coherent interconnects. *arXiv preprint arXiv:2409.08141* (2024).

[13] Thurlow, R. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 5531, May 2009.

[14] Xu, P., and Roscoe, T. The NIC should be part of the OS. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2025), HotOS '25, Association for Computing Machinery, p. 151–157.

A Appendix

A.1 Simple Adder

The following program demonstrates how to write a simple, non-nested handler, including serializers and deserializers for its inputs and outputs and how to register the handler as a service with the lauberhorn runtime. The handler is a simple adder program that takes in a `struct add_call` with two members `int32_t a`, `int32_t b` and returns a `struct add_resp` with a member `int32_t c = a + b`. For readability, we assume that all API calls succeed and therefore do not need any error checking.

Note: As we do not invoke the `add_handler` as a client here, we would not have to define `unmarshal_resp` and `marshal_call` in `add_ops`. However, the example code below in A.2 will make use of this which is why we already define it.

```c
#include <arpa/inet.h>
#include <stdlib.h>
#include "lauberhorn.h"

struct add_call {
    int32_t a;
    int32_t b;
};

struct add_resp {
    int32_t c;
};

void *add_handler_fn(void *data, void *msg, int xid) {
    struct add_call *args = msg;
    struct add_resp *resp = calloc(1, sizeof(struct add_resp));

    resp->c = args->a + args->b;
    return resp;
}

void add_result_free(void *data) {
    free(data);
}

int add_unmarshal_call(void *in_buf, void *out_buf, size_t in_buf_len, void
↪  *priv) {
    struct add_call *call = out_buf;

    call->a = ntohl(*(uint32_t *)in_buf);
    call->b = ntohl(*((uint32_t *)in_buf + 1));
    return 0;
}

int add_unmarshal_resp(void *in_buf, void *out_buf, size_t in_buf_len, void
↪  *priv) {
```

```
    struct add_resp *resp = out_buf;

    resp->c = ntohl(*(uint32_t *)in_buf);
    return 0;
}

int add_marshal_call(void *in_buf, void *out_buf, size_t in_buf_len, void *priv)
↪  {
    struct add_call *call = in_buf;

    *(uint32_t *)out_buf = htonl(call->a);
    *((uint32_t *)out_buf + 1) = htonl(call->b);
    return 2 * sizeof(uint32_t);
}

int add_marshal_resp(void *in_buf, void *out_buf, size_t in_buf_len, void *priv)
↪  {
    struct add_resp *resp = in_buf;

    *(uint32_t *)out_buf = htonl(resp->c);
    return sizeof(uint32_t);
}

void *add_codec_alloc(rpc_alloc_kind_t kind, void *priv) {
    switch (kind) {
        case RPC_ALLOC_CALL:
            return calloc(1, sizeof(struct add_call));
        case RPC_ALLOC_RESP:
            return calloc(1, sizeof(struct add_resp));
        default:
            return NULL;
    }
}

void add_codec_free(void *ptr, rpc_free_kind_t kind, void *priv) {
    free(ptr);
}

static rpc_ops_t add_ops = {
    .marshal_call   = add_marshal_call,
    .marshal_resp   = add_marshal_resp,
    .unmarshal_call = add_unmarshal_call,
    .unmarshal_resp = add_unmarshal_resp,
    .alloc          = add_codec_alloc,
    .free           = add_codec_free,
};

static rpc_codec_t add_codec = {
    .ops  = &add_ops,
    .priv = NULL,
};

int main(void) {
    lauberhorn_t laub;
    lauberhorn_init(&laub);

    lauberhorn_handler_t handler = {
        .data = NULL,
        .func = add_handler_fn,
```

```
        .free = add_result_free,
        .mode = EXEC_INLINE,
    };

    /* register adder on port 12345, ONC RPC endpoint (1, 1, 1) */
    lauberhorn_reg_srv(&laub, handler, 1, 1, 1, 12345, &add_codec);

    /* start 4 worker threads (assumes >= 4 worker cores) */
    lauberhorn_worker_t workers[4];

    for (int i = 0; i < 4; i++)
        workers[i] = lauberhorn_create_worker(&laub, NULL, NULL);

    for (int i = 0; i < 4; i++)
        lauberhorn_join_worker(&laub, workers[i]);

    lauberhorn_fini(&laub);
    return 0;
}
```

A.2 Nested Adder

The following code demonstrates how to write a handler that makes use of nested calls. The defined handler receives a struct with 4 integers, invokes the `simple_adder` endpoint defined in A.1 twice - once per pair - and returns the two results.

Note: We assume that we have access to the static data structure `add_codec` defined in A.1.

```
#include "lauberhorn.h"

struct nested_add_call {
    int32_t a_1;
    int32_t a_2;
    int32_t b_1;
    int32_t b_2;
};

struct nested_add_resp {
    int32_t c_1;
    int32_t c_2;
};

void *nested_add_handler(void *data, void *msg, int xid) {
    struct nested_add_call *nested_call = msg;
    struct nested_add_resp *nested_resp = calloc(1, sizeof(struct
    ↪  nested_add_resp));

    /* endpoint information of the simple adder */
    lauberhorn_rpc_endpoint_t add_ep = {
        .daddr    = "192.168.129.222",
        .dport    = 12345,
        .prog_num = 1,
```

```
        .prog_ver = 1,
        .proc_num = 1,
        /* statically defined in the simple adder */
        .rpc_codec = &add_codec,
    };

    /* prepare arguments for the nested calls */
    struct add_call c1 = { .a = nested_call->a_1, .b = nested_call->b_1 };
    struct add_call c2 = { .a = nested_call->a_2, .b = nested_call->b_2 };

    /* dispatch both nested calls */
    lauberhorn_future_t f1, f2;
    lauberhorn_call_async(&add_ep, &c1, sizeof(struct add_call), &f1);
    lauberhorn_call_async(&add_ep, &c2, sizeof(struct add_call), &f2);

    /* add both futures to the await set */
    FUTURE_SET(set, 2);
    future_set_add(&set, &f1, &f2);

    /* Wait for both calls to complete. The order does not matter, so both
     * futures go into the same set and we await any until the set is empty. */
    lauberhorn_future_t *res;
    while ((res = future_set_await_any(&set)) != NULL) {
        struct add_resp *resp = res->result.data;

        if (res == &f1) {
            nested_resp->c_1 = resp->c;
        } else {
            nested_resp->c_2 = resp->c;
        }
        lauberhorn_free_result(&res->result);
    }
    return nested_resp;
}
```

A.3 Dandelion Compositions

Depth 2

Input: Three 2x2 matrices A, B, C
Output: $(A \cdot A) \cdot (B \cdot B) + (C \cdot C)$

```
function matmul (mat_in) => (matmul_out);
function matmac (mat_A, mat_B, mat_C) => (matmac_out);
composition graph_2 (matA, matB, matC) => (matResult) {
    matmul(mat_in = all matA) => (mat1_out = matmul_out);
    matmul(mat_in = all matB) => (mat2_out = matmul_out);
    matmul(mat_in = all matC) => (mat3_out = matmul_out);
    matmac(mat_A = all mat1_out, mat_B = all mat2_out, mat_C = all mat3_out) =>
    ↪  (matResult = matmac_out);
}
```

Depth 4

Input: 2x2 matrix A
Output: $(A \cdot A)^8$

```
function matmul (mat_in) => (matmul_out);
```

```
composition graph_4 (matA) => (matResult) {
    matmul(mat_in = all matA) => (mat1_out = matmul_out);
    matmul(mat_in = all mat1_out) => (mat2_out = matmul_out);
    matmul(mat_in = all mat2_out) => (mat3_out = matmul_out);
    matmul(mat_in = all mat3_out) => (matResult = matmul_out);
}
```