Copper and Wire: Bridging Expressiveness and Performance for Service Mesh Policies

Divyanshu Saxena
dsaxena@cs.utexas.edu
The University of Texas at Austin
Austin, USA

William Zhang
wtzhang23@gmail.com
The University of Texas at Austin
Austin, USA

Shankara Pailoor
spailoor@cs.utexas.edu
The University of Texas at Austin
Austin, USA

Isil Dillig
isil@cs.utexas.edu
The University of Texas at Austin
Austin, USA

Aditya Akella
akella@cs.utexas.edu
The University of Texas at Austin
Austin, USA

Abstract

Distributed microservice applications require a convenient means of controlling L7 communication between services. Service meshes have emerged as a popular approach to achieving this. However, current service mesh frameworks are difficult to use – they burden developers in realizing even simple communication policies, lack compatibility with diverse dataplanes, and introduce performance and resource overheads. We identify the root causes of these drawbacks and propose a ground-up new mesh architecture that overcomes them. We develop novel abstractions for mesh communication, a new mesh policy language centered on these abstractions to enable expressive policies, and a novel control plane that enables using minimal dataplane resources for policy enforcement. We develop the precise semantics of our language abstractions and demonstrate how our control plane can use them to execute policies correctly and optimally. We build and evaluate a prototype on realistic workloads and policies and open-source production traces. Our results show that complex policies can be specified in up to 6.75× fewer lines, enforced with up to 2.6× smaller tail latencies and up to 39% fewer CPU resources than today.

CCS Concepts: • **Networks** → **Programming interfaces**; **Programmable networks**; **Network management**.

Keywords: Service Mesh, Microservices, Cloud Computing

ACM Reference Format:
Divyanshu Saxena, William Zhang, Shankara Pailoor, Isil Dillig, and Aditya Akella. 2025. Copper and Wire: Bridging Expressiveness and Performance for Service Mesh Policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3669940.3707257

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707257

1 Introduction

Organizations are increasingly migrating from monolithic applications to distributed microservice deployments [23]. In a microservice architecture, network communication is critical, necessitating policies for access control, telemetry, traffic management, authorization, etc. These policies can be complex, involving fine-grained control over layer 7 requests/responses and connections. However, incorporating them directly into the application complicates application design and deployment, especially amidst dynamic policy updates. *Service meshes* [4, 5, 8, 11, 32] have emerged to address this challenge by abstracting communication policies into separate *sidecar* containers deployed alongside each service container. The sidecars collectively form the dataplane for the service mesh. The mesh control plane receives developer policies and configures the dataplane.

Ideally, service meshes should allow developers to easily roll out rich policies matching the sophistication of their applications and deploy them with minimal overhead. Unfortunately, current mesh designs severely constrain developers.

First, it is difficult for developers to realize *expressive communication policies easily*. Today's mesh control- and dataplanes only allow control over requests targeted at a *service*, but microservice communication involves *request sequences*, where the processing of a request triggers another. Developers wishing to write policies for such sequences of requests may need to write multiple 'sub-policies' targeting different services to realize even simple policies (§2). The process may also involve manually identifying which outgoing request was triggered by an incoming request which today also necessitates application source modification (§2); to make matters worse, this may have to be repeated with every change to the application's structure, such as the addition of a microservice. In addition, today's control planes expose narrow controls over specific dataplane features, forcing developers to intricately understand the dataplane sidecar to realize richer policies. Furthermore, today's control plane policy interfaces

tightly couple control planes to the underlying dataplane designs making it challenging for developers to leverage rich features in *diverse dataplanes*. In particular, developers wanting to use multiple distinct dataplane proxies – e.g., to exploit the feature-performance tradeoffs between different dataplane sidecars – must use multiple control planes, one per dataplane. Similar issues arise when developers attempt to exploit newly-proposed dataplane features (§2).

Second, the *end-to-end overhead* imposed by service meshes is significant. We argue (§2) that this is architecturally rooted in control planes lacking information about the application communication and dataplane semantics. Together, these preclude informed placement of policies across sidecars. The resulting superfluous use of sidecars (§2) – which today amounts to deploying a sidecar configured with *all* mesh policies at *every* service– results in significant latency inflation and CPU/memory overheads.

We address these drawbacks via a principled re-architecting of service meshes with Copper, a new mesh policy language, and Wire, a new mesh control plane (Figure 1).

We introduce *abstract communication types* (ACTs), a novel abstraction to capture the entities of interest in microservice communication (e.g., requests, connections) and the actions that can be performed on them. ACTs *break the coupling between data and control planes* such that dataplane-specific details are transparent to policy specifications, freeing developers from dealing with low-level understanding of the dataplane. In our approach, dataplanes vendors expose their functionality through *interfaces*, where they can sub-type a small set of generic ACTs to create new ACTs, allowing them to define their own ACTs and actions. Developers can then use these interfaces as header files in their policies, enabling *policy expression for diverse, heterogeneous dataplanes*.

Copper achieves *rich policy specification* by tracking – and enabling policy expression over – *context patterns*. Copper policies operate on instances of ACTs (e.g., a specific request), each tagged with the sequence of events leading to its creation, which we call its *run-time context*. Policies are defined over *context patterns* encoded as regular expressions – thus eliding the exact event sequence and exposing only relevant parts for policy execution. This design simplifies policy writing, as developers don't need to write multiple policies for each service involved in the target sequence, and policies remain unaffected by changes to an application's microservice architecture. In order to track run-time contexts at low overhead, we further develop an eBPF-based dataplane add-on. We show how to implement this within the constraints of the eBPF verifier, imposing a mere 10μs overhead on request processing.

Wire is a performance-oriented control plane that enables *control over sidecar overheads* by using key aspects of policy and application semantics. To assist Wire, Copper's dataplane interfaces include simple annotations indicating where an ACT action can be correctly executed (§4.1.3). Wire uses these annotations, along with the application graph and ACT type/subtype relationships in a MaxSAT formulation to determine the fewest sidecars to deploy, which dataplanes to use, and the policies for each sidecar (§5).

We describe the precise semantics of the key components of our approach – ACTs, contexts, interfaces, and the Copper policy language. Using the semantics, we formally prove that our framework correctly enforces arbitrary developer-specified policies at minimal dataplane cost.

We evaluate a prototype of our framework using realistic workloads against today's approaches. Our approach finds a good sweet-spot between expressiveness, ease of use, and performance. We show that Copper policies are simpler, more intuitive to write, and up to 6.75× smaller than the policies required by existing meshes – several policies can be expressed in less than 10 lines in Copper! By systematically reducing sidecars, Wire can offer up to 2.6× smaller tail latencies and 3× higher throughput, and also yield up to 39% and 52% lower CPU and memory usage, respectively. We also evaluate the efficacy of Wire on production traces, demonstrating up to 64% fewer sidecars used.

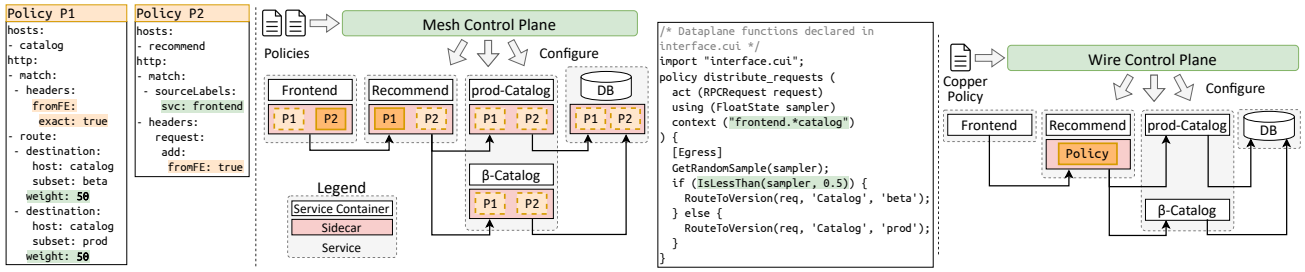
In summary, we make the following contributions:

- We propose ACTs that encapsulate the objects of interest in microservice networks, and run-time contexts that associate an ACT instance to an event sequence.
- We introduce Copper, a new mesh policy language that uses ACTs and contexts to simplify policy expression.
- To track run-time contexts at low overhead, we design an eBPF-based add-on to extend today's mesh dataplane.
- We design the Wire control plane that optimizes the dataplane to enforce Copper policies with minimal overheads.
- We evaluate our prototype on realistic workloads, demonstrating significant improvement over modern frameworks.

2 Background and Motivation

In a microservice architecture, each incoming request may invoke a subset of services via *cascading requests* (e.g., via RPCs) [23]. For large deployments, requests can traverse different subsets of services, resulting in complex communication patterns. To manage this, application developers commonly impose a variety of *policies*, such as for traffic management, telemetry, or access control. These policies often require *fine-grained* control over individual requests. For example, a service might want to deny only those requests that have a 'beta' header. Similarly, common deployment practices like canary releases and A/B testing necessitate routing requests to the appropriate service version.

Implementing these management features within application code increases developer burden and makes deployment and troubleshooting challenging. To address this, the service mesh was devised as a *separate* infrastructure layer that decouples application logic from communication policies [20].



(a) Workflow in current frameworks. **(b)** Workflow with Copper Wire.

Figure 1. Enforcing the policy: "Distribute requests from `Frontend` to the two versions of `Catalog` in a 50:50 ratio" in (a) Today's frameworks: Developers need to write multiple sub-policies for different services. Control planes deploy them using heavy sidecars at all services. (b) With Copper Wire: A single, simple, and intuitive policy that Wire can enforce using just 1 sidecar!

2.1 Current Service Mesh Frameworks

Service meshes are typically implemented by attaching *proxies* or *sidecars* (e.g., Envoy [2]) to each application pod. The proxy intercepts all traffic to and from the application, and executes *filters* to perform various functions such as routing, header manipulation, and security. Configuring these filters can be very tedious and intricate, therefore, service meshes comprise of a *control plane* that provides a high-level interface to configure the proxies. Popular control planes, such as Istio [4] and Cilium [8] provide a YAML-based API to the users to write policies, and then suitably configure the dataplane proxies. An example policy is shown in Figure 1a.

2.2 Drawbacks in current frameworks

We describe the workflow and **three key drawbacks** of today's service meshes using an example from [12] (Figure 1a). Suppose a developer wishes to impose the policy: "distribute requests from `Frontend` to the two versions (prod and beta) of `Catalog` in a 50:50 ratio". Note that this policy should apply to the requests from `Recommend` to `Catalog`, provided these requests are triggered by a prior request from `Frontend`.

1. Difficulty in realizing policies: State-of-the-art mesh control planes, like Istio [4], use *microservice endpoints* (i.e., `hosts` or `services`) as policy handles. Consequently, developers must break down their intended policy, writing separate per-service policies for traffic destined for different services.

In Figure 1a, the developer must specify policy P1 to distribute requests to `Catalog` between its prod and beta versions. However, just P1 is insufficient because the intent is to only apply to requests originating at `Frontend`. Since only `Recommend` sends requests to `Catalog`, the sidecar at `Recommend` must identify if an outgoing request to `Catalog` was triggered by a request from `Frontend` and then apply the policy. Existing dataplanes do not support this natively, so additional steps are needed, including modifying application logic:

I. Uniquely tag requests from `Frontend`*:* Write a policy to add a custom header `fromFE: true` to each request from `Frontend` to `Recommend`, as shown by Policy P2 in Figure 1a.

II. Map incoming requests at `Recommend` *to outgoing ones:* Since the requests from `Frontend` are terminated at `Recommend`, the developer must modify the service logic for `Recommend` to propagate the `fromFE` header from the incoming request to the new outgoing request to `Catalog`.

III. Specify the traffic distribution policy: At this point, Policy P1 can be applied to check *all* requests destined to `Catalog` whether they have the custom header and accordingly route to prod/beta versions. Since only requests from `Recommend` have the custom header this enforces the policy accurately.

These elaborate steps notwithstanding, developers struggle with policy specification as mesh control planes only offer narrow interfaces to common configuration knobs – e.g., `route`, `weight` parameters for routing policies (Figure 1a) – leaving the developer to configure other parameters of interest *directly at the dataplane*, which requires intricate knowledge (e.g., to configure rate limiting the developer has to directly interact with Envoy[1]). Similarly, stateful/-conditional policies (e.g., for sticky load-balancing[2]) are not exposed by control plane APIs despite dataplane support.

Overall, ad-hoc, narrow control plane APIs and coarse abstractions of service end-points as policy handles in existing control planes complicate policy specification; and, the lack of a dataplane mechanism to identify request sequences forces application source modifications.

2. Dataplane heterogeneity not well supported: Different dataplane sidecars offer varying performance-functionality tradeoffs. For instance, Istio proxy [4] is feature-rich but imposes a heavy performance penalty, while Cilium proxy [7] and Linkerd [5] are lightweight but only support a limited set of features. Developers may want to exploit these tradeoffs by using multiple dataplanes, e.g., using a lightweight dataplane for simple policies and a feature-rich one for complex policies. In today's mesh architectures, however, control

[1]This requires selecting appropriate rate limiter implementation, knowing how to add it to Envoy and then setting the specific (as many as 40!) parameters [16]

[2]Sticky load-balancing requires the requests of the same client to be routed to the same destination, irrespective of the load balancing weights

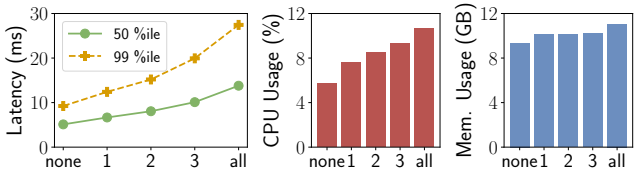


Figure 2. Overheads of mesh sidecars: We use a four-service chain from Hotel Reservation [23], and incrementally add sidecars to the services. X-axis shows the depth of the microservice graph until which sidecars are injected. For a workload using 100 requests per second, the 99 %-ile latency goes from 9.2ms to 27.5ms, while the CPU usage increases from 5.7% to 10.65% as we add sidecars.

plane interfaces are tied to specific dataplanes. As a result, developers wanting to use multiple dataplanes must manage multiple control planes, each with separate configurations. For a similar reason, dataplane upgrades are not well supported. New features added to the dataplane (e.g., to Envoy) need to be first upstreamed by control plane (e.g., Istio) implementers and then exposed to the policy writers via an updated API, which can take months.

3. Control plane constrains dataplane performance: Policies written by the developers are submitted to the mesh control plane (as shown in Figure 1a), which configures the sidecars accordingly. Current control planes are naive; e.g., they do not account for how an application's services communicate. Thus, they configure *each policy* on *all sidecars in the dataplane* – in turn, necessitating sidecars for *all services* (Figure 1a). This exhaustive use of sidecars in today's frameworks leads to significant dataplane overheads [31, 32, 34]. Each sidecar constantly occupies CPU and memory and adds significant latency due to L7 processing, e.g., parsing requests, executing policy actions, and forwarding requests.

These overheads increase with the number of sidecars in the system. We show a hop-by-hop analysis of sidecar overheads in Figure 2; the overheads increase as we incrementally add sidecars to the services of a microservice graph. Note that not all policies are executed at all sidecars; e.g., in Figure 1a, P1 and P2 are only executed at two of the sidecars (`Recommend` and `Frontend` - shown by the dark boxes). Further note that P1 just sets headers to requests from `Frontend` to `Recommend`, so applying P1 at the sidecar of `Recommend` instead of `Frontend`, can further reduce the sidecars needed to just 1. So, in theory, we could have reduced the overheads by deploying just a single sidecar instead of 5, still ensuring correct policy execution. However, current control planes fail to make this optimization as they are unaware of application communication and policy execution semantics.

2.3 Related Work

Mesh policy specification. Numerous industry solutions for service mesh policies exist, such as Istio [4], Cilium Mesh [8], NGINX mesh [11], Linkerd [5], Consul Connect [15], etc. However, these universally leverage coarse-grained policy handles and support only a dedicated dataplane. Approaches such as Rego [19] and [24], target "safety" policies by introducing control flow and service tree abstractions, respectively. Another work, AUTOARMOR [28], focuses on automatic generation of access control policies by extracting the request flow via static analysis of microservice code. However, these policy simplification techniques cannot be generalized to other service mesh needs, such as traffic management and telemetry. Additionally, they do not attempt to support multiple dataplanes.

Dataplane heterogeneity in service meshes. ServiceRouter [32] is one attempt at using multiple dataplanes in a single service mesh framework. It features a common RPC library for four different types of L7 routers, but suffers from the issue of being inextensible to new dataplanes for lack of common dataplane abstractions. It also requires intrusive application modification to link and use the RPC library.

Mesh overheads. MeshInsight [34] helps quantify and predict service mesh overheads whereas we focus on *avoiding* it. Existing solutions for managing mesh overheads, such as ServiceRouter [32], mRPC [21] and gRPC Proxyless [17], heavily depend on linking a custom RPC library into the application, which again is intrusive. It also breaks the transparent sidecar abstraction provided by service meshes.

3 An Overview of Our Approach

To address today's drawbacks we propose a new mesh architecture with a domain-specific language, Copper, and a co-designed, application-aware control plane, Wire. Our framework's salient aspects are (Figure 3):

A new abstraction for mesh communication. We propose *Abstract Communication Types*, or ACTs, (§4.1.1) that serve as an interface between policy writers and dataplanes, abstracting away low-level details of the dataplane and enabling dataplane-agnostic policies.

A clean interface to use dataplane features. Dataplanes provide their functionalities in the form of Copper *interface* files (§4.1.3). In Figure 3, step 1, dataplanes `d1` and `d2` express that they can perform `SetDeadline` and `SetHeader` operations, respectively. Developers can use these interfaces as headers in Copper policies.

Contexts. Our framework allows programmers to write policies over *contexts* (highlighted text in step 2 in Figure 3) expressed using regex patterns (§4.2). Each policy operates on concrete ACT instances, which we call "communication objects" (§4.1.2). Contexts allow rich encoding of service interactions using a succinct regex representation.

An application- and policy-aware control plane. Wire uses application graph, coupled with dataplane semantics from Copper interfaces and policy information from Copper programs, to place policies across a minimal number of sidecars, optimizing the performance of the mesh dataplane (§5).

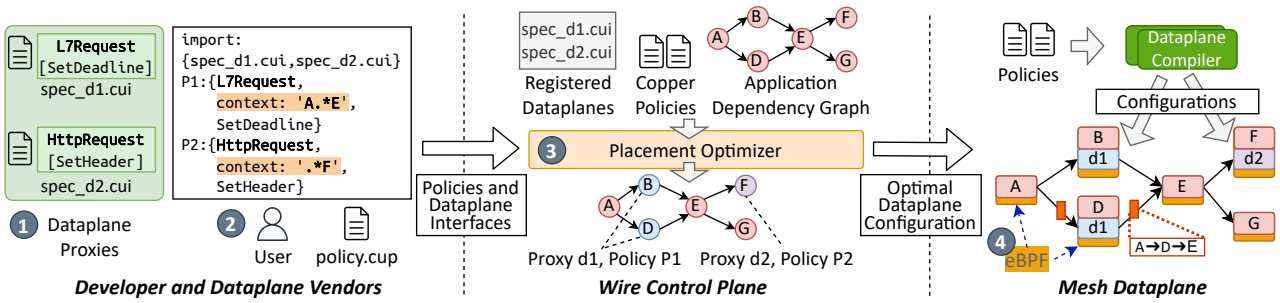


Figure 3. Overview of our approach. ①: Dataplane vendors provide interface files for their dataplanes. ②: Developers use these interface files to write Copper programs – P1 applies to requests from `A` to `E`, while policy P2 applies to all requests to `F`. ③: Wire uses the application graph and Copper programs to place policies across a minimal number of sidecars. Note that the `SetDeadline` function can only be executed on the sender service, and is hence, executed on sidecars of services `B` and `D`, instead of being executed simply at `E`. ④: An eBPF add-on tracks contexts in the datapath – the context `A`⟶`D`⟶`E` implies the request from `D` to `E` was triggered by a request from `A` to `D`.

```
act Request {
   action Deny(self)
   action GetHeader(self, header_name),
   action SetHeader(self, header_name, header_value),
}
act Response {
   action GetStatusCode(self),
   action GetHeader(self, header_name),
   action SetHeader(self, header_name, header_value)
}
act Connection {
   action SetTimeout(self, timeout),
   action SetMaxOpenConnections(self, max_conn)
}
```

Listing 1. Generic ACTs in Copper.

In Figure 3, Wire identifies that three sidecars are sufficient to implement the policies (step 3).

A lightweight dataplane add-on to track contexts. Enforcing such high-level policies requires the dataplane to track the contexts. By default, a mesh architecture would need to rely on sidecars at all services to propagate such contexts, but this adds overhead. We avoid this by tracking and propagating contexts in the datapath via a lightweight and efficient eBPF-based dataplane add-on (§6), attached to each service pod (step 4 in Figure 3).

4 The Copper Language

Copper is a high-level mesh language that enables writing rich policies and supports diverse and evolving dataplanes.

4.1 Copper Building Blocks

4.1.1 Abstract Communication Types. An Abstract Communication Type (ACT) encapsulates objects of interest to microservice communication. Copper defines three *generic* ACTs to represent commonly used network objects in meshes – requests, responses, or connections – and common actions on them (see Listing 1). We intentionally keep the actions on generic ACTs simple and limited in number, so that all dataplanes can support them. As we show in §4.2, this is crucial for enabling dataplane-agnostic policies.

Copper allows ACTs to be subtyped so that dataplane vendors can define their own *derived* ACTs and specify supported actions on them, without worrying about the availability of those features in other dataplanes or about the control plane lifting them. For example, the `Connection` ACT can be subtyped by a dataplane to define a `TCPConnection` ACT with *additional* actions such as `SetTCPKeepAlive`, `SetTCPNoDelay`, etc. Similarly, the `Request` class can derive various L7 request types, such as `HTTPRequest` and `gRPCRequest`.

An ACT can be implemented in different ways by dataplane vendors as long as they provide the functionality exposed through the interface. For functions defined in the generic ACTs (e.g., `GetHeader`), implementations across different dataplanes are expected to produce the same outputs.

Mesh policies operate over instances of ACTs, which we refer to as *communication objects*, or COs for short. In a microservice application, the creation of COs is triggered by events. For instance, in our example from Figure 1a, a request CO from `Recommend` to `Catalog` is triggered when `Recommend` processes another request CO from `Frontend`.

4.1.2 Run-time Contexts. To support fine-grained control over COs, Copper associates each CO with a *run-time context* that captures the sequence of events that led to the creation of the object. Formally, we represent a CO as $o = (\tau, \sigma)$, where τ is the type of the object (e.g. `Request`, `Response`, `Connection`, or any of the derived ACT types) and σ is a sequence of events $[e_1, \ldots, e_n]$ that triggered the creation of o. We refer to σ as the run-time context of o and represent events as a triple (s_i, a_i, d_i), where s_i is the source service that created the CO a_i for destination service d_i; we also denote s_i and d_i with $\mathbb{S}(a_i)$ and $\mathbb{D}(a_i)$, respectively.

The event sequence is causal: for consecutive events (s_i, a_i, d_i) and $(s_{i+1}, a_{i+1}, d_{i+1})$, the destination service d_i of an earlier event becomes the source for the next event ($d_i = s_{i+1}$), and a_{i+1} is created as a consequence of d_i receiving a_i. Figure 4 shows an example of run-time contexts for cascading requests. Current dataplanes do not track run-time contexts – to this end, we use a lightweight eBPF add-on (§6).

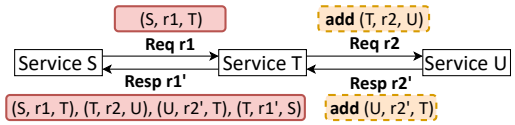


Figure 4. Run time contexts of COs: Solid boxes show the run-time contexts and the dashed boxes show how the context gets modified.

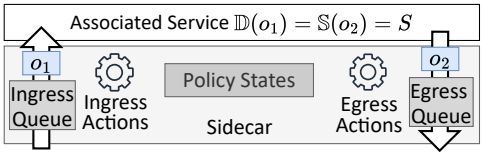


Figure 5. Sidecar model.

```
import common.cui;
state FloatState{
   action GetRandomSample(self),
   action IsLessThan(self, float value),
}
act RPCRequest: Request{
   action GetHeader(self, string header_name),
   action SetHeader(self, string header_name, string value),
   action Deny(self),

   [Egress]
   action RouteToVersion(self, string service, string label),
}
```

Listing 2. An example dataplane interface that declares a `RPCRequest` ACT that subtypes the generic `Request` ACT and a state `FloatState` to operate over floating point numbers.

4.1.3 Copper Dataplane Interfaces.

Before describing our final building block – Copper interfaces – we present the *abstract sidecar model* assumed by Copper and leveraged in its interfaces (Figure 5): For a CO o_1 directed to a service S, the CO is received via the ingress queue at the sidecar and the resulting CO from S, o_2, is sent via the egress queue of the sidecar. When a CO is at the head of the ingress (egress) queue, the sidecar can execute *actions* on it, called *ingress* (*egress*) actions. These actions may modify the CO or the sidecar state. We model the *sidecar state* as a triple $S = (\Sigma, I, E)$ where I and E denote the ingress and egress queues respectively, and Σ corresponds to some sidecar-local state. Note that a CO o can only be operated upon by egress actions on the egress queue at its source service $\mathbb{S}(o)$. Similarly, a CO o can only be operated upon by ingress actions on the ingress queue at its destination service $\mathbb{D}(o)$ (from the definitions of $\mathbb{S}(o)$ and $\mathbb{D}(o)$ in §4.1.2).

Returning to the final building block, dataplane vendors expose their supported functionality through Copper *interfaces*. A Copper interface must describe the ACTs the dataplane supports and the supported *state types* that policy writers can use to maintain state during policy execution.

Figure 6 presents the grammar for Copper interfaces, which consists of a list of ACT and state type definitions. Each ACT is declared using the notation `act T1:T` indicating that the dataplane supports the `T1` ACT, which is a subtype of `T`. Actions supported by the dataplane are indicated using the

Copper Interfaces:

$\langle annotation \rangle$ ::= `Ingress` | `Egress`
$\langle action \rangle$ ::= $(\langle annotation \rangle, \langle action\_name \rangle, \langle arg \rangle^*)$
$\langle act \rangle$ ::= $(\langle act \rangle?, \langle action \rangle+)$
$\langle state\_action \rangle$ ::= $(\langle action\_name \rangle, \langle arg \rangle^*)$
$\langle state \rangle$ ::= $(\langle state\_action \rangle+)$
$\langle interface \rangle$ ::= $(\langle interface\_name \rangle, (\langle act \rangle \mid \langle state \rangle)+)$

Copper Programs:

$\langle expr \rangle$ ::= $\langle action \rangle$
| $\langle state\_action \rangle$
| `if` $\langle expr \rangle$ $\langle expr \rangle$ $\langle expr \rangle$
$\langle section \rangle$ ::= $(\langle annotation \rangle, \langle expr \rangle+)$
$\langle policy \rangle$ ::= $(\langle interface \rangle+, \langle act \rangle, \langle context \rangle, \langle state \rangle?, \langle section \rangle+)$

Figure 6. Grammar for Copper interfaces and policies.

`action` keyword. State types are similarly specified by the `state` keyword along with the operations that can be performed on the state. Listing 2 shows an example interface.

Using the ACT abstraction allows dataplanes to transparently expose their functionalities to developers, without relying on often incomplete and narrow control plane APIs. As dataplanes evolve, the updated features can also be directly reflected in the dataplane interface, without waiting for a control plane to expose them.

Action Annotations Copper requires dataplane vendors to *annotate* their actions using two keywords - `[Egress]` and `[Ingress]`. These annotations provide information on whether the action semantics *restrict* it to run on the egress queue or the ingress queue. For instance, the `RouteToVersion` action on an RPC (Listing 2) can only be executed on the client side of the RPC (when the request is being sent), and hence, the egress queue of the sidecar. Given our sidecar model, an action annotated by `Egress` is only well-defined for a CO o in the egress queue of the sidecar at $\mathbb{S}(o)$. Similarly, action annotated by `Ingress` is only well-defined for a CO o in the ingress queue of the sidecar at $\mathbb{D}(o)$.

Actions without any annotations (e.g. `Deny` in Listing 2) are assumed to be able to run on *either* ingress or egress queues. For actions with both annotations, it is assumed that the action should execute on a CO at the egress queue of its source service $\mathbb{S}(o)$ *and* the ingress queue of its destination service $\mathbb{D}(o)$. The annotations are very simple but general enough to specify the correct execution semantics for the generic ACTs, and by extension, also for any derived ACTs. We show the power of these annotations in §5 where they help the control plane to optimize data plane performance.

4.2 Copper Policy Programs

Copper *policy* programs enable programmers to use the dataplane interfaces to write policies on COs.

Figure 6 shows the syntax of Copper policies, and Listing 3 shows an example Copper policy. Every policy consists of four parts: (i) the `act` annotation to specify the type of the CO that this policy applies to, (ii) the `using` annotation to specify what state the policy maintains, (iii) the `context` keyword to specify a regular expression over the contexts

of COs on which the policy should apply (referred to as *context pattern*), and (iv) the body of the policy to specify the actions that should be performed on the CO. These policies can be then compiled using a dataplane-provided compiler. Notably, Copper policies allow the use of conditionals and program-local states, enhancing the language's expressiveness. Copper can support more complex use cases—such as maintaining persistent states across COs for sticky load-balancing; however, the dataplane compiler is responsible for ensuring the correctness of the underlying actions.

Policy semantics Copper policies are executed on dataplane sidecars. A Copper policy takes as input a CO o, that can be on the ingress or egress queue of that sidecar (Figure 5). Each Copper policy has two separate sections annotated as `[Egress]` and `[Ingress]`, either of which can be empty but not both at the same time. If o's run-time context belongs to the set of strings represented by the policy `context` (see formal definition below), then the relevant portion of the policy body is executed – if the communication object is on the Ingress (resp. Egress) queue, then the statements found in the Ingress (resp. Egress) section are executed.

Formally, Copper policies can be represented as a 4-tuple $(\mathcal{T}, C, A_E, A_I)$, where $\mathcal{T}$ specifies the target ACT, C is a context, expressed as a *regular expression*, and A_E and A_I are sequences of actions that execute on the CO when it is in the egress and ingress queues of the sidecar, respectively. We say that a policy $\pi = (\mathcal{T}, C, A_E, A_I)$ *matches* a CO $o = (\tau, \sigma)$ where $\sigma = [(s_1, a_1, s_2), (s_2, a_2, s_3) \ldots (s_n, a_n, s_{n+1})]$, iff (1) τ is a subtype of $\mathcal{T}$, and (2) the string $s_1 s_2, \ldots s_{n+1}$ is accepted by the regular expression C.

Valid Context Patterns The sidecar model (Figure 5) dictates that A_I must only execute on a matching CO o at the ingress queue of $\mathbb{D}(o)$, and A_E must only execute on a matching CO o at the egress queue of $\mathbb{S}(o)$. Hence, context patterns that do not provide a unique source or destination service are considered invalid. Thus, a valid context pattern in Copper must have the form '$C'S$.' or '$C'S$', where C' is a general regular expression. The former pertains to COs with $\mathbb{S}(o) = S$ while the latter denotes COs with $\mathbb{D}(o) = S$. The only exception to this rule is the context pattern '*', which matches all COs, and can be used to specify mesh-wide policies.

Example Copper Policy Consider the policy from Figure 1 of routing requests from `Frontend` to `Catalog`. The programmer can specify the context `"Frontend.*Catalog"` to match all requests of interest. Listing 3 shows how this can be written in Copper using the interface from Listing 2. Since the interface annotates the `RouteToVersion` action as `Egress`, we can use it only in the egress section of the policy.

Using regex context patterns simplifies policy writing by (i) reducing the number of policies needed—for instance, one policy for a context pattern versus multiple policies for each service, and (ii) enabling policies to be written independently of changes to the microservice architecture.

```
import "interface.cui";
policy route_requests (
   act (RPCRequest request)
   using (FloatState sampler)
   context ('Frontend.*Catalog')
) {
   [Egress]
   GetRandomSample(sampler);
   if (IsLessThan(sampler, 0.5)) {
      RouteToVersion(request, 'Catalog', 'beta');
   } else {
      RouteToVersion(request, 'Catalog', 'prod');
   }
}
```

Listing 3. An Example Policy Program

```
/* fast_interface.cui */
import "common.cui";
act L7Request: Request{
   action Deny(self),
   action GetHeader(self, string header_name),
   action SetHeader(self, string header_name, string value),
}

/* policy2.cup */
import "interface.cui";
import "fast_interface.cui";
policy checkout_headers (
   act (Request req)
   context ('Checkout'.'Catalog')
) {
   [Ingress]
   SetHeader(req, 'low-priority', 'true');
}
```

Listing 4. Dataplane-agnostic Policy Example

Copper's generic ACTs allow programmers to write policies independent of the underlying dataplane implementations, enabling users to leverage different dataplanes transparently. In Figure 1, suppose a new `Checkout` service is added that can send requests to the `Catalog` service. As it is a newly added service, developers want to tag all requests from `Checkout` with a `low-priority` header. They have at their disposal two dataplanes: `Proxy1` with interface in Listing 2 and a faster alternative, `Proxy2` with interface in Listing 4. With Copper, the developers can write a generic policy (Listing 4) that uses the `SetHeader` action on the generic `Request` ACT (instead of specifying `RPCRequest` or `L7Request` from `Proxy1`/`Proxy2`'s interfaces). The control plane then can choose `Proxy2` if no other policies require the `RouteToVersion` action (only supported by `Proxy1`).

5 The Wire Control Plane

Wire is a mesh control plane that uses action semantics encoded in Copper interfaces to minimize the sidecar overhead.

Wire takes as input an application graph G, a set of dataplanes T, a set of Copper policies Π and returns an optimal policy placement. Below, we define the graph and optimal policy placement, and then describe a solver-aided technique for finding such a placement.

For a microservice application, the *application graph* is a directed graph $G(V, E)$ where the nodes V are the services and an edge $(u, v) \in E$ indicates that u can send a CO to v directly. Such graphs are easy to collect [28], and have been

used for various purposes in microservice deployments, such as autoscaling [33], performance analysis [25] and access control policy generation [28].

A *policy placement* is a mapping Γ from services $s \in V$ to tuples (T_s, Π_s) where $T_s \in T$ is the sidecar to be attached to s and $\Pi_s \subseteq \Pi$ are the policies that must run on the sidecar of s. A valid placement assigns policies to sidecars such that every CO that *should* be processed by a policy *will* be processed. Formally, a valid placement must assign a policy $\pi = (\mathcal{T}, C, A_E, A_I)$ to the sidecar of service $\mathbb{S}(o)$ if $A_E \neq \emptyset$ and to $\mathbb{D}(o)$ if $A_I \neq \emptyset$ for every matching CO o.

We assume the user provides a cost $c \geq 0$ for each sidecar in T. An *optimal policy placement* Γ^{Opt} is a valid policy placement that minimizes $\Sigma_{s \in \Gamma} C(T_s)$.

MaxSAT Reduction We now describe how Wire leverages the information encoded in policies' context patterns and dataplane interfaces to reduce the optimal placement problem to an instance of weighted MaxSAT [27]. MaxSAT aims to satisfy some hard constraints while maximizing the combined weight of satisfied soft constraints. We elide the full details of our MaxSAT encoding for brevity, and only describe the intuition and the key constraints here.

For a policy $\pi = (\mathcal{T}, C, A_E, A_I)$, Copper constrains the context pattern C to be of the form $C'S$ or $C'S(.)$ (see §4.2) – any CO o matching π must either have source service $\mathbb{S}(o) = S$ or destination service $\mathbb{D}(o) = S$. Thus, the actions A_E must execute at the set of source services of all matching COs (denoted $\mathbb{S}_\pi$), and the actions A_I must execute at the set of destination services of all matching COs (denoted $\mathbb{D}_\pi$). Note that $\mathbb{S}_\pi$ and $\mathbb{D}_\pi$ can be easily computed using the context pattern C and the graph G. For contexts of the form $C'S(.)$, $\mathbb{S}_\pi$ is simply $\{S\}$ and for contexts of the form $C'S$, it is the subset of in-neighbors of S that are consistent with the context C'. Similar rules can be derived for $\mathbb{D}_\pi$ as well.

However, oftentimes the actions specified under A_E can also be run under A_I (and vice-versa). In particular, actions without any annotations in the dataplane interface fall under this category (see §4.1.3). Such actions only modify the CO without any side-effects on the policy state at the sidecar, e.g. `GetHeader`, `SetHeader`, etc. We refer to policies with such actions as *free-policies*; for a matching CO o for such a policy, the actions A_E (A_I) can be safely executed at the ingress (egress) queue at $\mathbb{D}(o)$ ($\mathbb{S}(o)$) as well.

Finally, a policy π may use actions that restrict the set of sidecars that can support it (denoted T_π). Thus, if π is assigned to a service s, the sidecar at s must be one of T_π. Using the above observations, Wire generates three types of hard constraints, described formally below. We use the boolean $p_{i,j}$ to indicate that policy π_i is placed on a sidecar attached to service s_j and the boolean $q_{k,j}$ to indicate that sidecar T_k is attached to s_j.

1. *Policy placement constraints:* For policy $\pi_i = (\mathcal{T}, C, A_E, A_I)$ that is not a free-policy, then the egress section of the policy, A_E, must execute at all possible source services, i.e., the set $\mathbb{S}_{\pi_i}$. Similarly, the ingress section of the policy, A_I, must execute at all possible destination services, i.e., the set $\mathbb{D}_{\pi_i}$.

$$\wedge_{s_j \in \mathbb{S}_{\pi_i} \cup \mathbb{D}_{\pi_i}} p_{i,j} \text{ if } \pi_i \text{ is not a free-policy} \tag{1}$$

2. *Free-policy constraints:* For a free-policy $\pi_i = (\mathcal{T}, C, A_E, A_I)$, both sections of the policy can be executed either at all services in $\mathbb{S}_{\pi_i}$ or at all services in $\mathbb{D}_{\pi_i}$.

$$\wedge_{s_j \in \mathbb{S}_{\pi_i}} p_{i,j} \oplus \wedge_{s_j \in \mathbb{D}_{\pi_i}} p_{i,j} \text{ if } \pi_i \text{ is a free-policy} \tag{2}$$

3. *Sidecar placement constraints:* Given a policy π, at most one sidecar can be attached to a service and that the sidecar can only be one of T_π.

$$p_{i,j} \rightarrow \exists k : (q_{k,j} \wedge \neg q_{k',j} \forall k' \neq k) \tag{3}$$

$$p_{i,j} \rightarrow \oplus_{k \in T_\pi} q_{k,j} \tag{4}$$

Finally, Wire also takes a static cost model to assign each sidecar T_s a cost $C(T_s) \geq 0$. Application owners may assign different costs to available dataplanes to reflect different priorities - for example, assigning a higher cost for a heavy dataplane to avoid its performance overheads or a lower cost to a more reliable dataplane. Given this cost function, Wire adds the clause $\neg q_{j,k}$ as a soft constraint for each service j and sidecar T_k and assigns it the weight $C(T_k)$. We add a cost for $\neg q_{j,k}$ because minimizing the total cost of sidecars placed is equivalent to maximizing cost of sidecars *not placed*.

We provide the generated encoding to a MaxSAT solver, which produces assignments for: (i) where policies should be executed, (ii) which sidecars are deployed at which services, and (iii) for which policies the actions A_E (A_I) should be executed on the Ingress (Egress). Using (iii), Wire re-writes free policies by moving the A_E (A_I) actions to Ingress (Egress) section of the policy, to construct the set of updated policies, Π'. It is for this updated set Π' for which our solution is both valid and optimal, as stated by the following theorem:

Theorem 1 (Correctness)**.** *Given an application graph G and a set of policies Π, let Γ denote the policy placement generated by Wire, along with optimized policies Π'. Then Γ is a valid and optimal solution with respect to G and policies Π'.*

Proof Sketch. Our proof relies on the concept of a *valid placement set* for a policy π. We say a set of services S_π is a valid placement set for π if every CO that should be processed by π is processed by π at one of the services in S_π. We call S_π minimal if no subset $S' \subset S_\pi$ is a valid placement set for π. Note that for free policies, there can be multiple valid placement sets. Using this definition, we can prove the above theorem in three steps.

First, we can prove by contradiction that that a optimal policy placement Γ^{Opt} must include a *minimal valid placement set* for each policy $\pi \in \Pi$. Intuitively, if that is not the case, then we can find a subset S' of the set of services Γ^{Opt} assigns π to, that is also a valid placement set for π – contradicting the optimality of Γ^{Opt}.

Next, we can show that the generated constraints exhaust all valid placement sets for all policies by exhausting all possible combinations of context patterns (whether it is $C'S(.)$ or $C'S$) and policy types (whether it is free policy or not). The above analysis proves that any solution to the MaxSAT reduction will result in a valid placement set for each policy.

Since the MaxSAT maximizes the combined weight of the satisfied soft clauses (constraints Eq. 3 and Eq. 4), its output is the placement that minimizes the total cost of sidecars placed. Hence, the final solution to the MaxSAT encoding must be an optimal policy placement.

6 Dataplane Context Propagation

To enforce policies, Copper dataplanes need to operate on COs' run-time contexts. However, COs do not carry the context information by default. In our example from Figure 1a, the CO from `Recommend` to `Catalog` does not carry the context to relate it to the CO from `Frontend` to `Recommend` that triggered it. Hence, to apply policies over contexts, we need a mechanism to add these run-time contexts to the COs.

To do so, we need to identify that two COs are related. This can be done by correlating the COs via a unique *trace ID* header that is propagated in the service calls (also known as context propagation [3]). Next, we need to associate the traceID of the outgoing request with the correct run-time context, and do so without modifying applications. A layman approach uses sidecar proxies to read and update context in requests, but this adds overhead by requiring sidecars at all services. Instead, we propose using eBPF [1] for transparent context propagation in the dataplane.

The challenge is this requires parsing of L7 headers (e.g., gRPC [3] uses HTTP/2), which are typically compressed via encoding [26] or indexing [30]. Such complex processing is not amenable to eBPF verifier restrictions and may increase latencies due to memory copy overhead with eBPF maps during lookups. We overcome this challenge using two clever ideas (Figure 7). First, we avoid stateful processing overhead by directly looking for the encoded `traceID` header instead of parsing each header. Second, we add the raw bytes of the context in outgoing requests as a new `CTX` HTTP/2 frame instead of encoding them in HTTP/2 headers – vastly simplifying our eBPF programs.

In order to avoid false positives when looking up the context in the `ctx_map`, we use `traceID` header of a request as the key (Figure 7) – `traceID` is generated by tracing libraries to uniquely identify different requests, making collisions very rare. To further reduce the probability of collisions, once a request exits a service, its `traceID` is removed from `ctx_map`.

Another challenge is that for arbitrary service names, the contexts can become very long. This is problematic because eBPF programs in kernel currently limit the stack usage to just 512B. Additionally, arbitrarily long contexts can add overheads as they make request headers very bulky. Hence, we encode each service with a service identifier and only tag each CO with the context string $s_1 s_2 \ldots s_n$, used for policy execution (§4.2). This allows us to support contexts of up to 100 services, enough to satisfy 96% of requests in production traces [29]. However, this is not a fundamental limitation, and with advances in eBPF-kernel programming supporting bigger stack sizes, this limit can be easily increased.

We implement this dataplane add-on using cgroup socket hooks. These hooks allow us to attach separate eBPF programs for sockets specific to a cgroup, ensuring isolation between service containers. They also facilitate direct processing of L7 payloads, independent of lower TCP/IP layer concerns such as packet drops or out-of-order packets. We employ four eBPF programs for path propagation (Table 1).

[3]Context propagation is a standard practice for tracing and monitoring in microservice applications, done using distributed tracing libraries [6].

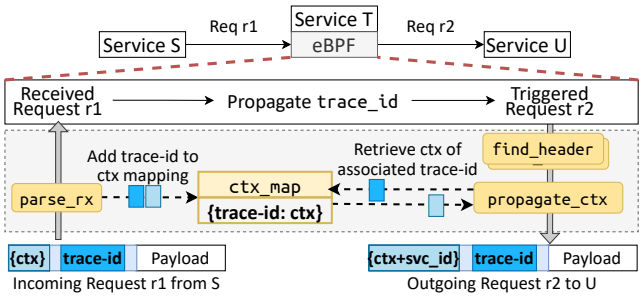


Figure 7. Leveraging eBPF hooks to propagate the run-time context. The figure expands on the context propagation for service `T`, for a request `r1` sent from service `S` that triggers a request `r2` to service `U`. Each request shows the additional `CTX` frame and `trace-id` headers.

Program (Attach Hook)	Description
`add_socket` (`sockops`)	Track all open sockets on a service container. (Below three programs are attached to these sockets).
`parse_rx` (`sk_skb`)	Parse the incoming request to extract `traceID` and `context` fields and save them in `context_map`.
`find_header` (`sk_msg`)	Search for `traceID` header in the outgoing request and make a *tail call* to `propagate_ctx`.
`propagate_ctx` (`sk_msg`)	Lookup `traceID` of outgoing request in `context_map` to get the context; add it to the outgoing request.

Table 1. Details of the used eBPF programs.

Application	Workload	Services
Boutique **(OB)** [12]	Index Page	10
Hotel Reservation **(HR)** [23]	Mixed Workload (25% for each of search, recommend, user and reserve queries)	18
Social Network **(SN)** [23]	Mixed Workload (60% for timelines, 30% for users and 10% for posts)	26

Table 2. Experiment details for the three benchmark applications.

7 Evaluation

We implement a parser and compiler for Copper (6K LOC in Rust), a prototype for the Wire control plane (2.2K Golang LOC), where we implement policy placement (§5), and an eBPF dataplane add-on (1.2K LOC). We perused GitHub repos [14, 18], Slack channels for popular mesh frameworks,

Policy Description	Target Service Sequences	App	Istio		Copper	
			Policy Lines (Δ SLoC)	Parameters	Policy Lines	Arguments
★ **P1:** Set 'display' header 'true' for all requests to `catalog` originating from `frontend`.	(frontend, catalog), (frontend, checkout, catalog), (frontend, recommend, catalog)	OB	54 (8 lines in 2 services)	13	8 (**6.75×**)	3 (**4.33×**)
P1: Set the 'critical' header to 'true' for all requests to `geo` and `rate` originating from `frontend`.	(frontend, search, geo), (frontend, search, rate)	HR	37 (4 lines in 1 service)	9	8 (**4.63×**)	3 (**3×**)
P1: Set 'write' header 'true' for all requests to `post-storage` originating from `compose-post`.	(compose-post, post-storage), (compose-post, user-timeline, post-storage), (compose-post, home-timeline, post-storage)	SN	54 (8 lines in 2 services)	13	8 (**6.75×**)	3 (**4.33×**)
P2: Route to v2 of a service if request is from `checkout`; v1 if from `frontend`	(frontend, cart), (frontend, currency), (frontend, catalog), (frontend, ship), (frontend, checkout, cart), (frontend, checkout, currency), (frontend, checkout, catalog), (frontend, checkout, ship)	OB	101 (4 lines in 1 service)	28	36 (**2.8×**)	12 (**2.33×**)
P2: Route to v2 of a service if request is from `search`; v1 if from `frontend`	(frontend, geo), (frontend, rate), (frontend, search, geo), (frontend, search, rate)	HR	59 (4 lines in 1 service)	16	18 (**3.28×**)	6 (**2.67×**)
★ **P2:** Route to v2 of a service if request is from `compose-post`; v1 if from `frontend`	(frontend, home-timeline), (frontend, user), (frontend, user-timeline), (frontend, compose-post, home-timeline), (frontend, compose-post, user), (frontend, compose-post, user-timeline)	SN	80 (12 lines in 3 services)	22	27 (**2.96×**)	9 (**2.44×**)
P3: Restrict access to database services	(cart, redis-cache)	OB	24	3	9 (**2.6×**)	3 (**1×**)
★ **P3:** Restrict access to database services	(reserve, mongo), (reserve, memcached), (profile, mongo), (profile, memcached), (geo, mongo), (rate, mongo), (rate, memcached), (recommend, mongo), (user, mongo)	HR	99	24	57 (**1.7×**)	24 (**1×**)
P3: Restrict access to database services	(user, mongo), (user, memcached), (social-graph, mongo), (social-graph, redis), (url, mongo), (url, memcached), (post-storage, mongo), (post-storage, memcached), (user-timeline, redis), (user-timeline, mongo), (user-mention, mongo), (user-mention, memcached)	SN	99	24	60 (**1.65×**)	24 (**1×**)
★ **P4:** Rate limit requests from `frontend` to `catalog`	(frontend, catalog), (frontend, checkout, catalog), (frontend, recommend, catalog)	OB	92 (8 lines in 2 services)	35	16 (**5.75×**)	9 (**3.89×**)

Table 3. Study of representative policies on benchmark applications when using Istio vs Copper. The table shows the service chains targeted by each policy, lines of code (LoC) and source lines of code (SLoC) changes needed for Istio and the corresponding lines and arguments needed in Copper (with improvement over Istio shown in brackets) – Copper requires 1.65-6.75× fewer lines and 1-4.33× fewer arguments. Star-marked policies are pictorially depicted in Figure 8.

and Istio's documentation on common service mesh tasks [9], and discussed with mesh operators to identify four categories of common policies. In what follows, we pick example policies for each category and evaluate them against popular microservice benchmarks to answer these questions:

1. Does Copper help enable simple and expressive mesh policies relative to today's approaches? (§7.1)
2. How beneficial is Wire for real-world applications in lowering dataplane overhead and enabling the effective use of multiple dataplanes compared to today's best approaches? And, how scalable is Wire? (§7.2)
3. What are the overheads of using the eBPF add-on? (§7.3)

7.1 Evaluating Copper

In this section, we will demonstrate how using ACTs, contexts and interfaces make Copper policies simpler and intuitive. Our four policy categories - access control, traffic management, header manipulation - are shown in Table 3, along with (a) examples of each for the benchmark applications and service sequences to which those policies apply, (b) relevant application subgraphs for a benchmark for each policy category (Figure 8), and (c) the corresponding Copper program for these subgraphs (Listings 5–8).

[P1] ***Header Manipulation*** P1 modifies the header for *all* requests to a service that originated at `frontend`; i.e., the same

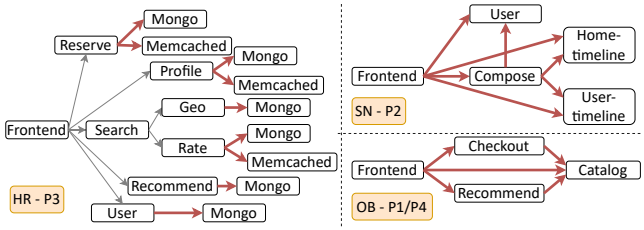


Figure 8. Application subgraphs for star-marked policies in Table 3. Bold arrows indicate relevant requests for the policies.

```
1  import "interface.cui";
2  policy catalog_write (
3     act (RPCRequest request)
4     context ("frontendservice.*productcatalogservice")
5  ) {
6     [Ingress]
7     SetHeader(request, "display", "true");
8  }
```

Listing 5. P1 policy for OB benchmark

policy is applied to all service sequences shown in Table 3. Using Istio, developers would need to write separate policies for each sequence, but in ***Copper, programmers can specify all sequences via a single context pattern*** (see Listing 5, line 4 as an example), yielding 4.63–6.75× fewer policy LoC, without source modifications.

[P2] ***Traffic Management*** P2 routes traffic to version v1 of a service if the request came directly from the `frontend`,

```
1  import "interface.cui";
2  policy social_route (
3     act (RPCRequest request)
4     context ("frontend.*home-timeline")
5  ) {
6     [Egress]
7     if (GetContext(request) == "frontendhome-timeline") {
8        RouteToVersion(request, "home-timeline-service", "v1");
9     } else {
10       RouteToVersion(request, "home-timeline-service", "v2");
11    }
12 }
```

Listing 6. P2 policy for SN benchmark (Similar programs for two other contexts – see Table 3)

```
1  import "interface.cui";
2  policy accessor_reservation (
3     act (RPCRequest request)
4     context ('rate.')
5  ) {
6     [Egress]
7     Allow(request, 'rate', 'mongo-rate');
8     Allow(request, 'rate', 'memcached-rate');
9  }
```

Listing 7. P3 policy for HR benchmark (Similar programs for five other services – see Table 3)

```
1  import "interface.cui";
2  policy count_social (
3     act (RPCRequest request)
4     using (Counter counter, Timer timer)
5     context ("frontendservice.*productcatalogservice")
6  ) {
7     [Ingress]
8     Increment(counter);
9     if (IsTimeSince(timer, 60)) {
10       if (IsGreaterThan(counter, 1000)) {
11          Deny(request);
12       }
13       Reset(timer);
14       Reset(counter);
15    }
16 }
```

Listing 8. P4 policy for OB benchmark

and to version v2 if it came from another intermediate service (see Table 3 and Figure 8). Similar to Figure 1a, in Istio, this requires configuring and modifying intermediate services (`checkout` for OB, `search` for HR, and `compose-post` for SN). However, ***Copper can encode intermediate services in context patterns*** (see Listing 6, line 4), leading to 2.8–3.28× fewer policy LoC, without source modifications.

[P3] *Simple Access Control* P3 applies access control to restrict access to each database service in the application, requiring setting of 'allow' rules for each accessor-database pair (shown in Table 3). Since these policies apply to service pairs, Copper and Istio require same number of arguments. ***Copper's high-level programming abstractions make the policies more concise*** (Listing 7), leading to 1.65-2.6× fewer lines compared to Istio's verbose YAML configurations.

[P4] *Rate Limiting* P4 applies a rate limit on the requests from `frontend` to `catalog`. Istio does not expose an API to use Envoy's rate-limiting feature (§2) – hence, the programmer must write a low-level Envoy filter, requiring several non-trivial parameters to be set as well as knowledge of Envoy's architecture. ***ACTs in Copper hide such low-level details*** (see Listing 8); combined with high-level constructs like conditionals (§4.2), Copper can express P4 in 5.75× fewer LOC than Istio.

7.2 Evaluating Wire

In this section, we will demonstrate how the knowledge of application graphs and policy semantics (to know whether a policy is a free policy (§5)), leads to improved dataplane performance. We will also show that Wire can further improve the performance by leveraging multiple dataplanes, made possible by the use of generic ACTs and Copper interfaces.

7.2.1 Evaluation on Benchmark Applications. We deploy the three benchmarks ("OB", "HR", "SN" in Table 2) on an 80-core Cloudlab [22] cluster (4X 20-core Intel Xeon CPU@2.40GHz) with 64GB of RAM and connected via 10Gbps Ethernet. We use another identical machine to generate load, using the `wrk2` [13] benchmarking tool with 10 threads.

Methodology We use two baselines:

1. Current control planes (Istio): Use Istio's control plane that deploys Istio-proxy at all services.

2. Optimized control planes (Istio++): Augment Istio with the knowledge of application graphs to remove sidecars from services where no policy is enforced.

We configure the above baselines and Wire for the policy classes described in §7.1. P3 would result in the same sidecar deployment as P1, as both are free policies (§4.1.3). Similarly, P4 would result in the same deployment as P2, as both their actions can only be executed on `Egress`. Thus, we only present the results using P1 and P2 below.

For a comprehensive evaluation, we extend the policies listed in Table 3 to include *all possible contexts* originating from the frontend service in each benchmark application. To demonstrate Wire's capability of leveraging multiple dataplanes, we also evaluate a combination of P1 and P2; thus, in effect, we evaluate the following two policies:

Policy P1. *Set header for requests originating at* `frontend`. Since database services typically do not perform header processing, we only apply this policy to non-database services. We add header manipulation rules to requests originating at the frontend service per benchmark.

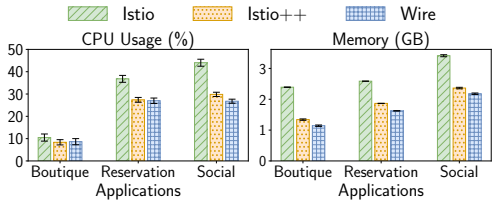
Policy P1+P2. *Set header for requests originating at* `frontend` *and route to version v1.* We apply P1, like before, on non-database services but P2 on all services. Since the benchmarks only have one version for each service, for testing, we implement this policy by configuring load-balancing on the sidecars, with a weight of 100% to a single version.

We use two mesh dataplanes for our experiments – Istio-proxy [10], a feature-rich but bulky dataplane, and Cilium-proxy [7], a lightweight alternative with limited features. P1 can only be enforced by Istio-proxy as Cilium-proxy does not support header-manipulation. P2, on the other hand, can be enforced by both dataplanes, Istio-proxy and Cilium-proxy. For each dataplane, we profile the sidecar and assign a cost based on its overhead on the 99%-ile latency.

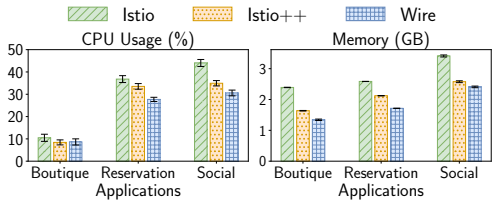
(a) Policy P1

(b) Policy P1+P2

Figure 9. Tail Latencies against applied rate.



(a) Policy P1



(b) Policy P1+P2

Figure 10. CPU and Memory at operating throughput.

Latency and Throughput Figure 9 shows the 99%-ile tail latencies of the applications versus applied load. For P1 (Figure 9a), ***Wire can consistently support higher throughput (request rate) compared to both baselines, while ensuring lower tail latencies***. Wire's supported request rate is 1.67-3× and 1-1.25× higher than Istio and Istio++, respectively, across the three benchmarks. At low loads, Wire's tail latencies are up to 2.6× and 1.9× lower than Istio and Istio++. For P2 (Figure 9b), ***Wire can provide better performance against both baselines*** - Wire can support 1.33-2.33× and 1.25-1.87× higher throughputs compared to Istio and Istio++, respectively, across the three benchmarks. As we will discuss below, the benefits against Istio++ increase in P2 because of Wire's capability to leverage multiple dataplanes.

CPU and Memory Usage Because Wire can reduce the number of sidecars and, wherever possible, replace heavy Istio-proxy with Cilium-proxy, it can lower CPU and memory usage. Figure 10 shows that across the two policies, using Wire results in 2-39% lower CPU usage and 7-52% smaller memory usage against the baselines. Wire's resource benefits are greater for larger microservice graphs – 2-17% lesser CPU for OB versus 10-39% lesser for SN, as there is more room for reducing the number of sidecars.

Key Takeaways We now note key observations from the above results and delineate the underlying reasons. To elucidate the observed benefits, we also provide the configurations of sidecars for P1 and P1+P2 in Figure 11. The figure shows where each of the baselines and Wire deploy Istio sidecars for the two policies.

Even with a single dataplane, Wire can provide improvements by using application graphs and free policy semantics. For P1, all three control planes are forced to use Istio-proxy as Cilium-proxy doesn't support header manipulation. However, Istio deploys Istio-proxy at all services as it does not know application dependencies or policy semantics – leading

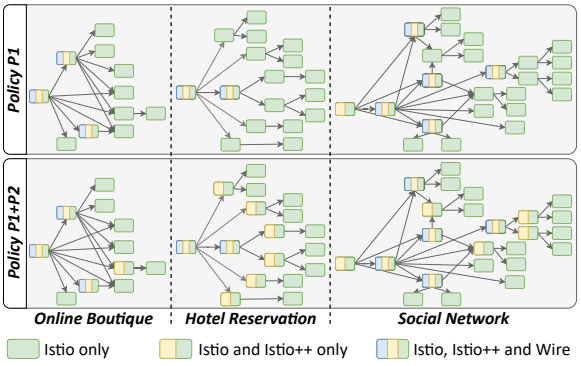


Figure 11. Application graphs showing where each control plane applies heavyweight Istio sidecars for policies P1 and P1+P2. Single colored boxes show services where only Istio applies a sidecar, double-colored show where Istio and Istio++ apply, and three-colored boxes show where all three control planes apply sidecars.

to 10, 18, and 26 sidecars for the OB, HR and SN benchmarks respectively (see Figure 11). With the Istio++ baseline, a few sidecars can be avoided as the set-header policy can be enforced by running only at services that call other non-database services. Thus Istio++ uses only 3, 2, and 6 Istio-proxies for OB, HR, and SN, respectively (Figure 11). However, since the set-header policy is a *free-policy* (§5). Wire uses this along with the graph to further optimize placement and to deploy 3, 2, and 5 sidecars respectively for OB, HR, and SN (Figure 11). This is the same as Istio++ for OB and HR (hence the observed similar performance), but 1 fewer sidecar for SN. In particular, Wire can avoid the sidecar at the `frontend` service in SN – leading to less queueing at the 'hot-spot' `frontend` service and yielding greater improvements than the other two benchmarks.

Even when policies are constrained to run at Ingress/Egress, Wire can efficiently leverage multiple dataplanes. For P1+P2, the Istio baseline deploys sidecars at all services (similar

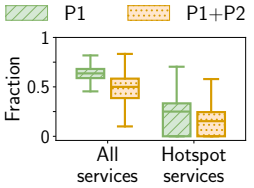


Figure 12. Fraction of services without sidecars for policies P1 and P1+P2.

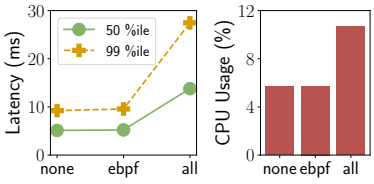


Figure 13. Overheads of eBPF add-on at all services vs sidecar at all (represented as 'all').

to P1). However, because the routing policy applies to all possible contexts, in Istio++, sidecars are also needed at all intermediate services, to propagate contexts (similar to the example in Figure 1a). Thus, Istio++ deploys at all non-leaf services in the graph – 4, 8, and 10 sidecars for OB, HR and SN respectively (see Figure 11). With Wire, since the eBPF add-on natively provides path propagation, we can still deploy the same number of Istio-proxies as needed for P1 and only Cilium-proxies for P2 – thus providing significant improvements over both Istio and Istio++.

7.2.2 Evaluation on Production Traces. We evaluate the efficacy of the Wire control plane using a production trace from Alibaba [29], containing service call traces for several microservice applications. We take the top 750 most popular applications and use their call traces to construct the application graphs. We use the policy sets P1 and P1+P2 from §7.2.1 and test Wire on each (application, policy) pair, assuming a single dataplane available for use. Figure 12 shows that the median fraction of services without sidecars is 0.64 and 0.5 for P1 and P1+P2, respectively. The figure also shows that on average, Wire can avoid sidecars from 22% and 15% of all hotspot services (services with more than 4 edges in the graph) for P1 and P2, respectively. This is crucial because we find that 30% of all requests are for such hotspot services.

7.2.3 Scalability of Wire. For the benchmark applications, Wire can find the optimal placement for the policy sets P1 and P2 in less than 50ms. For the top 750 applications from the production traces with application graphs ranging from 24–329 services and 37–892 edges, Wire finds the optimal placement in 565ms on average, and with a maximum of 9.8s, across all graphs and the two policies.

7.3 eBPF Add-on Overheads

To evaluate the end-to-end overhead of eBPF, we repeat the experiment from Figure 2. Our findings are shown in Figure 13 – attaching the eBPF add-on at all services, results in a small end-to-end overhead of 90μs on median latency and 240μs on 99p latency. Note that this is significantly lower than the overheads imposed by sidecars (causing ~3× worse tail latencies). Figure 13 also shows that the CPU overhead of tracking context in eBPF is negligible. To further understand the per-hop overhead, we run a gRPC echo server microbenchmark. We run the server in a Kubernetes pod, with the eBPF add-on attached. We then run multi-threaded clients, with 4-32 threads, to send requests to the pod. We find that the overheads imposed by the eBPF add-on are small and constant across the number of clients - average per-hop latency is inflated by just 8μs. We test the eBPF add-on with larger context sizes – even for the maximum supported context length in our prototype of 100, the overheads are below 10μs per-hop. In contrast, sidecars add roughly 1-3ms of overhead per hop (see Figure 2, also in [32, 34]).

8 Concluding Remarks

We introduced a new service mesh framework with new control and dataplane building blocks that significantly improve programming ease and policy enforcement overhead. We offer a few concluding remarks about our approach.

Policies that don't benefit from Wire Wire's performance improvements arise from its capability to optimize the placement of free-policies (§5) - hence, if all policies submitted to Wire are non-free, Wire will not be able to remove sidecars. A mesh use case for such a policy is mTLS authentication over service exchanges. Even in such cases, Wire can still optimize dataplanes by choosing lightweight sidecars at services that only require mTLS and heavier ones where complex policy enforcement is needed.

Copper does not restrict inter-service communication mechanism Enforcing Copper policies only relies on the context being carried in the request - hence, the inter-service communication mechanism does not affect policy enforcement. However, the eBPF add-on (§6) must be modified as per the protocol to propagate the context. Our prototype considers gRPC-type communication that uses HTTP/2, but can be easily extended to Thrift RPCs, message queues, etc.

Migration efforts for existing frameworks To add a new dataplane to Copper, vendors must create an interface file (with appropriate action annotations) and a compiler to translate actions into low-level filter configurations. Note that since the dataplane vendors are the ones who implement specific features in the dataplane, they can decide if the implementation would support the feature on both the ingress and egress queues or not – hence, for most actions it should be straightforward to decide the action annotations. With Copper, the burden of annotating actions and compiling user policies is offloaded to the dataplanes. We believe this is a necessary overhead to enable a common control plane and encourage future innovations by allowing vendors to create their own compilers.

Conflicting Policies It is possible for Copper policies to conflict – for instance, a `RouteToVersion()` action being applied to a request that is also `Deny()`-ed. These conflicts can occur in current mesh frameworks as well, and resolving them is a challenging but interesting future direction. We believe that the proposed ACT abstractions and action annotations can be handy tools in tackling this.

Acknowledgements

We thank our shepherd, Aastha Mehta, and the anonymous reviewers for their insightful comments and the members of UTNS Lab for the regular discussions and feedback. This research was supported by NSF Grants CNS-2214015 and CNS-2023222.

A Artifact Appendix

A.1 Abstract

This artifact includes the source code for the implementation of the proposed system. It constitutes of two major components: the Copper programming language and the Wire control plane. It also includes the scripts required to evaluate the system on a Cloudlab cluster. We open-source this artifact to allow other researchers and developers to use and improve it in their own work. In this appendix, we briefly describe the steps to set up and run the cluster, and to reproduce the results. More detailed instructions can be found in the README files in the respective repositories.

A.2 Artifact check-list (meta-information)

- **Program:** We use the Social Network and Hotel Reservation applications from the DeathStarBench [23] benchmark and the Online Boutique application [12]
- **Run-time environment:** Our artifact was built and evaluated on Ubuntu 20.04, running Linux version 5.15. The Linux version is important as the eBPF add-on was built for this version.
- **Hardware:** Our system does not require any specific hardware to run but our evaluation scripts are developed for Cloudlab clusters.
- **Output:** Evaluation of our system results in log files that can be analyzed to get the numbers.
- **Experiments:** The experiments show the benefits of using Copper (via LoC counts) and Wire (via latency and load measurements) over baselines. Our evaluation was run over Cloudlab.
- **How much disk space required (approximately)?:** Nearly 20GB of disk space is needed to run the benchmarks, and up to 100MB to store the results.
- **How much time is needed to prepare workflow (approximately)?:** Preparing the workflow requires running a few commands that run for up to an hour.
- **How much time is needed to complete experiments (approximately)?:** We run a 60 second workload for several request rates for three microservice benchmarks, each repeated for all baselines. This can take up to 18 hours to run.
- **Publicly available?:** The codebase is available at https://github.com/utnslab/copperlang for the language implementation and at https://github.com/utnslab/wire-mesh for the Wire control plane.
- **Code licenses (if publicly available)?:** Our code is available under the MIT License.
- **Archived (provide DOI)?:** The Copper language implementation is archived at https://doi.org/10.5281/zenodo.14053526 and the Wire control plane is archived at https://doi.org/10.5281/zenodo.14053530. Both are also available on GitHub, where the code is maintained.

A.3 Description

A.3.1 How to access. The artifact is publicly available on GitHub at https://github.com/utnslab/copperlang and https://github.com/utnslab/wire-mesh. This repository contains all of the source code and links (submodules) to dependencies. The repository is also archived on Zenodo and can be accessed at https://doi.org/10.5281/zenodo.14053530.

A.3.2 Hardware dependencies. While Copper-Wire itself are not tied to any particular hardware architecture, our evaluation was run on xl170 Cloudlab nodes. We expect similar trends to be observed on other Cloudlab nodes as well.

A.3.3 Software dependencies. Our artifact was built and evaluated on Ubuntu 20.04, running Linux version 5.15. The Linux version is important as the eBPF add-on was built for this version. Running the Copper language parser requires Rust version 1.78.0.

A.4 Installation

The README.md in the repositories lists the detailed steps to setup Wire mesh. Copper implementation only requires Rust version 1.78.0 or above – no other setup is needed. A brief summary of setting up Wire mesh is provided below.

1. Clone the repositories:
```
git clone https://github.com/utnslab/copperlang
git clone https://github.com/utnslab/wire-mesh
cd wire-mesh
git submodule update -init -recursive
```
2. Start a 5-node Cloudlab cluster with the `small-lan` profile.
3. Setup the Cloudlab cluster (this requires setting environment variables):
```
export CLOUDLAB_USERNAME=<username>
export CLOUDLAB_PROJECT=<project>
export CLOUDLAB_CLUSTER=<cluster>
cd wire-mesh
./cloudlab/config.sh <exp_name> 0 3 0
./cloudlab/client_config.sh <exp_name> 4
```
where `exp_name` is the name of the cloudlab experiment, `username` is your Cloudlab username (found at https://www.cloudlab.us/myaccount.php), `project` is the Cloudlab project under which you are running the experiment, and `cluster` is the full domain of the specific Cloudlab cluster (e.g. `utah.cloudlab.us`). The above will start a 4-node kubernetes cluster with node 0 as the control node.

4. Check that the cluster is up and running: On *Cloudlab node 0*, run `kubectl get nodes` and check that all nodes are in the Ready state.

A.5 Experiment workflow

A.5.1 Evaluating Copper. Run:

```
cargo run --bin copper-generate --features="generate-bin"
<path to .cup file>
```

A.5.2 Evaluating Wire.

1. Start the respective microservice application: On *Cloudlab node 0*:
   ```
   cd wire-mesh/scripts/deployment
   cd <application (reservation/social/boutique)>
   ./deploy_p1.sh <(istio/hypothetical/wire)>
   ```
2. On your local node, run:
   ```
   ./cloudlab/run_experiment.sh <exp_name> <appl>
   <local_dir> <workload_rate>
   ```
   where `exp_name` is the name of the Cloudlab experiment and `appl` is one of `reservation, social or boutique`.

A.6 Evaluation and expected results

The evaluation scripts provided with the repository will generate graphs similar to the ones in the paper. There could be slight variations due to hardware differences.

Results can vary significantly for some reasons:

1. Kubernetes cluster is not set up properly.
2. The service mesh framework (Istio/Cilium) are not installed correctly.

References

[1] [n. d.]. eBPF. https://ebpf.io/.
[2] [n. d.]. Envoy Proxy. https://www.envoyproxy.io/.
[3] [n. d.]. gRPC: High-Performance RPC Framework. https://grpc.io/.
[4] [n. d.]. Istio. https://istio.io/.
[5] [n. d.]. Linkerd. https://linkerd.io/.
[6] [n. d.]. OpenTelemetry. https://opentelemetry.io/.
[7] 2023. Cilium Proxy. https://github.com/cilium/proxy.
[8] 2023. Cilium Service Mesh. https://docs.cilium.io/en/v1.13/network/servicemesh/index.html.
[9] 2023. Istio Tasks - Documentation. https://istio.io/latest/docs/tasks/.
[10] 2023. istio/proxy. https://github.com/istio/proxy.
[11] 2023. NGINX Service Mesh. https://www.nginx.com/products/nginx-service-mesh.
[12] 2023. Online Boutique Microservices Demo Application. https://github.com/GoogleCloudPlatform/microservices-demo.
[13] 2023. wrk2. https://github.com/giltene/wrk2.
[14] 2024. Cilium Github Repo. https://github.com/cilium/cilium.
[15] 2024. Cilium Service Mesh. https://developer.hashicorp.com/consul/docs/connect.
[16] 2024. Configuration of Envoy Rate-limiting in Istio. https://github.com/istio/istio/blob/f434191be1877d6aa4af01b5e6caec8c344c82bc/samples/ratelimit/local-rate-limit-service.yaml.
[17] 2024. gRPC Proxyless Service Mesh. https://cloud.google.com/traffic-director/docs/proxyless-overview.
[18] 2024. istio/istio. https://github.com/istio/istio.
[19] 2024. Open Policy Agent - Rego. https://www.openpolicyagent.org/docs/latest/policy-language/.
[20] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks* (Virtual Event, United Kingdom) *(HotNets '21)*. Association for Computing Machinery, New York, NY, USA, 229–236. https://doi.org/10.1145/3484266.3487379
[21] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote Procedure Call as a Managed System Service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 141–159. https://www.usenix.org/conference/nsdi23/presentation/chen-jingrong
[22] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19
[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013
[24] Karuna Grewal, P. Brighten Godfrey, and Justin Hsu. 2023. Expressive Policies For Microservice Networks. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 280–286. https://doi.org/10.1145/3626111.3628181
[25] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. 2023. Murphy: Performance Diagnosis of Distributed Cloud Applications. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 438–451. https://doi.org/10.1145/3603269.3604877
[26] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898
[27] Chu Min Li and Felip Manyà. 2021. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*. https://api.semanticscholar.org/CorpusID:28884712
[28] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. 2021. Automatic Policy Generation for Inter-Service Access Control of Microservices. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3971–3988. https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing
[29] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 412–426. https://doi.org/10.1145/3472883.3487003
[30] Roberto Peon and Herve Ruellan. 2015. HPACK: Header Compression for HTTP/2. RFC 7541. https://doi.org/10.17487/RFC7541
[31] Prateek Sahu, Lucy Zheng, Marco Bueso, Shijia Wei, Neeraja J. Yadwadkar, and Mohit Tiwari. 2023. Sidecars on the Central Lane: Impact of Network Proxies on Microservices. arXiv:2306.15792 [cs.DC]
[32] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal,

and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 969–985. https://www.usenix.org/conference/osdi23/presentation/saokar

[33] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 167–181. https://doi.org/10.1145/3445814.3446693

[34] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2022. Dissecting Service Mesh Overheads. https://doi.org/10.48550/ARXIV.2207.00592