Occam: A Programming System for Reliable Network Management

Jiarong Xing
Rice University

Kuo-Feng Hsu
Meta

Yiting Xia
Max Planck Institute for Informatics

Yan Cai
Meta

Yanping Li
Meta

Ying Zhang
Meta

Ang Chen
University of Michigan

Abstract

The complexity of large networks makes their management a daunting task. State-of-the-art network management tools use workflow systems for automation, but they do not adequately address the substantial challenges in operation reliability. This paper presents Occam, a programming system that simplifies the development of reliable network management tasks. We leverage the fact that most modern network management systems are backed with a source-of-truth database, and thus customize database techniques to the context of network management. Occam exposes an easy-to-use programming model for network operators to express the key management logic, while shielding them from reliability concerns, such as operational conflicts and task atomicity. Instead, the Occam runtime provides these reliability guardrails automatically. Our evaluation demonstrates Occam's effectiveness in simplifying management tasks, minimizing network vulnerable time and assisting with failure recovery.

CCS Concepts: • **Networks** → **Network management**; **Network reliability**; **Network manageability**.

Keywords: Network management, Reliability

ACM Reference Format:
Jiarong Xing, Kuo-Feng Hsu, Yiting Xia, Yan Cai, Yanping Li, Ying Zhang, and Ang Chen. 2024. Occam: A Programming System for Reliable Network Management. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627703.3650086

Xing and Hsu contributed equally to this work.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3650086

1 Introduction

Managing complex networks is challenging, particularly at scale. A planet-scale cloud/content provider's network can contain devices from dozens of vendors, dozens of network element roles, and tens of thousands of circuits in operation. Together, they contribute to thousands of configuration changes every day [25]. Furthermore, network management encompasses a wide range of tasks, including service upgrades, device updates, network expansion, and feature deployment. Each task requires a distinct workflow of operations. The combination of network scale and task diversity introduces inherent risks, turning network management into a complex and precarious procedure [20]. Thus, the network management system plays an important role for reliability.

Researchers and practitioners have made concerted efforts toward increasing the reliability of network management tasks. In the past years, our community has moved away from the use of "method of procedures" (MOPs)—loose text documenting management steps and rules of thumb. Traditional MOPs can be easily misinterpreted by individual operators and often require manual translation into scripts or configlets [17]. Rather, recent systems like AT&T's CORNET [28], Google's ZTN (zero-touch networks) [25], and Alibaba's NetCraft [27] utilize *workflow systems* to streamline management tasks and minimize manual efforts. A workflow program is general-purpose code (e.g., in Python or Java) written in a pipeline of stages. Each stage performs several execution steps and passes intermediate results to the next, and it can further invoke pre-defined sub-procedures called "building blocks" (BBs) for common network operations (e.g., drain traffic). BBs may be implemented in any shape or form, such as Ansible, Python, or CLI scripts, or even using precompiled binaries. Thus, the resulting workflow systems are akin to the UNIX Shell—powerful, general-purpose administration tools—applicable to any automation tasks, but without dedicated designs for network management.

While zero-touch automation is a laudable step forward, reliable management goes beyond automation. Just like how rogue Shell scripts may wipe an entire disk, a problematic workflow program can exert arbitrary influence to the network. Network management requires end-to-end coordination of myriad services (e.g., querying the source-of-truth

database [29, 39]) and devices (e.g., switch firmware updates), so reliability goals are essential:

- **Consistency**. A management task may require changes to multiple configurations on a device or across devices, often in a specific order. A workflow system should guarantee that changes are made in a consistent manner, resulting in correct network states.
- **Efficiency**. In many cases, it is desirable for changes to be deployed as quickly as possible, so as to mitigate failures, balance traffic, or roll out a security patch. Thus, scheduling and executing workflows efficiently to minimize network vulnerable time is important.
- **Resilience**. Workflows may fail for various reasons, e.g., wrong device states, conflicts with other workflows, failure of manual steps (such as maintenance in the field). Partially-executed workflows should be undone using rollback logic, and this is often not just a simple reversal of the original management steps.

Occam addresses these problems by developing a programming system that provides *reliability guardrails* for network management tasks. Our key observation is that network management tasks essentially *evolve the logical network state* from snapshot to snapshot, and then *modify the physical devices based on the logical state*. Further, production networks often store the logical network state in a source-of-truth database, such as at Facebook/Meta's Robotron [39] and Google's Malt [29]. Network management tasks, therefore, can be viewed as transactions atop a logical network database, with corresponding physical effects on the network devices. Thus, we can abstract the network management workflows as changes to the network state and apply various database techniques for reliable operations.

Occam exposes a uniform and constrained programming model using a *network object* abstraction as well as a set of APIs that manipulates the objects. An Occam program constructs network objects by scoping a range of devices and performs device operations through the APIs. The programming model shields complexities that arise from having to reason about reliability goals in a manual and task-specific manner, so that network operators can author concise and easy-to-use management tasks. Based on the network objects, the runtime system automatically generates queries to the source-of-truth database, enforces constraints on the network, monitors task progress, and suggests failure rollback plans. For consistency, Occam builds an *object tree* from network objects based on their dependencies and applies *multi-granularity locking* to enable workflow-level transactions. For efficiency, Occam performs *hierarchical lock scheduling* on the object tree to maximize task parallelism and minimize execution time. For resilience, Occam identifies limitations of reverse-order rollbacks and generates rollback plans by pattern matching based on the semantics of management operations. Our contributions include:

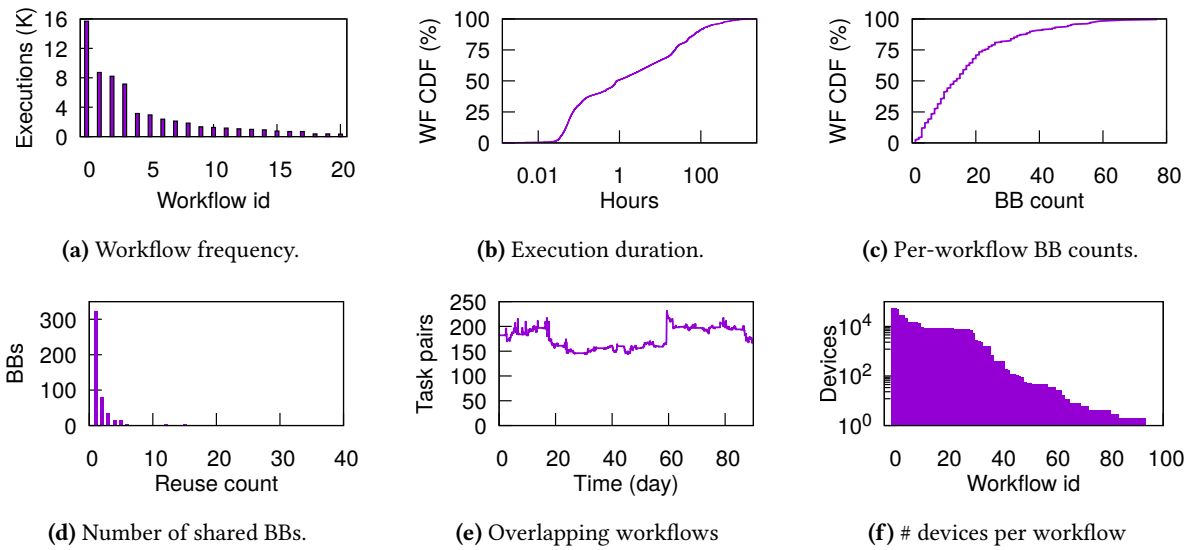
- An analysis of an industry dataset obtained from Meta, a major cloud/content provider, to understand the current practice and to motivate the Occam design (§2).
- A novel programming abstraction (§3)—a logically centralized *network object*—that simplifies the management tasks by shielding peripheral reliability concerns.
- A runtime system that ensures state consistency and atomicity (§4), schedules for execution efficiency (§5), and assists runtime failure and recovery (§6).
- A prototype (§7) and its evaluation (§8) to demonstrate its effectiveness for network management: the programming model reduces LoC of workflow programs by over 90%; locking and scheduling reduces task execution time by 4-10×; and Occam generates effective rollback plans to assist with recovery.
- Network management is a problem space that is today primarily worked on by industry. We contribute our prototype as evaluation tools for future research [42].

2 Insights from a Production Network

Conventional wisdom in network management circulates within industry operation teams. Although several aspects of the current practice have been published (e.g., data modeling [29, 39], automation [25, 27, 28], state management [38]), system support for network management remains largely opaque to the academic community. To gain insights into today's industry practice, we have engaged with Meta, a major online service provider that operates networks at scale. We were given anonymized datasets collected from their current management system over several months, supplemented with numerous interviews with their operation teams to understand the current practices. This section contributes a detailed analysis of today's industry practices. Occam builds upon the insights drawn from large network operations, but rethinks the design requirements for a clean-state architecture from an academic perspective.

2.1 Network management practices

The practice of relying on "method of procedures" (MOPs) [11, 12] was prevalent (in Meta and elsewhere) until very recently. MOPs are collections of documents written in natural languages that specify step-by-step procedures for management tasks—e.g., steps to drain traffic from a network. MOPs specify in a loose manner actions to be taken and network conditions to monitor. It is then up to the network operator to carry out the operation, e.g., either manually or by translating the MOPs into scripts or configlets. Ambiguity in natural languages and loose documentation make this a challenging task. Even across operation teams in the same company, scripts and configlets may be written in different languages without a uniform execution platform. Idiosyncratic coding practices introduce further obscurity. This disarray has motivated their recent transition to "workflow" automation.



(a) Workflow frequency. **(b)** Execution duration. **(c)** Per-workflow BB counts.

(d) Number of shared BBs. **(e)** Overlapping workflows **(f)** # devices per workflow

Figure 1. Statistics from Meta.

The Meta workflow system exhibits marked similarity to a recently-published project, AT&T's CORNET [28]. For both Meta and AT&T, the workflow systems are general-purpose automation frameworks, akin to Apache Airflow [18]. A more powerful analogue to Shell, they are administrative tools that orchestrate smaller building blocks (BBs) to perform larger tasks. In CORNET, BBs are scripts and configlets (e.g., Ansible, Python, or CLI code) to perform a specific network management step. In fact, any executable format would suffice, since the workflow system simply invokes them as a blackbox—just like how Shell strings together multiple arbitrary binaries in a script. A CORNET workflow is a pipeline of stages, where each stage embeds one or more BBs.

Occam's system has slightly more uniformity in that it is a purely Python-based platform, with the same language used for the main management programs and the individual BBs. This helps code reuse, maintenance, and understanding. However, the same workflow system is also in charge of many other tasks that are far removed from network management. Cloud storage backup, service repair, data encryption, and many such generic maintenance tasks use the workflow platform. Network management tasks, despite their domain-specific nature, are treated as just another automation workload. In other words, the underlying system is a legacy, general-purpose design with little customization for network management. Transitioning management tasks to this platform is, unfortunately, more of an afterthought.

2.2 Workflow characterization

Atop the workflow system, network management tasks are highly (but not completely) automated. Notable examples that require manual intervention are tasks that require physical setup (e.g., network expansion) or failure recovery (e.g., recovering crashed devices or services).

Our dataset identifies 234 workflow programs for network management, among which 118 (~50%) have been executed at least once in the measured duration. Moreover, there is a heavy-tail distribution for their execution frequency, as shown in Figure 1a. (The figure omits the long tail after the top-20 workflows for clarity of presentation.) The most frequently executed workflow ran more than 15,000 times over the month (or every three minutes on average), and about ten workflows executed more than 1000 times (or 1.4 times per hour). We found that these workflows are usually monitoring tasks, which oversee the network health and gather the latest device and network state.

Figure 1b further plots the task execution time for all workflow runs. Roughly over half of the executions lasted for more than one hour, and about 20% for more than 100 hours. The execution time is bottlenecked by two main factors: a) some workflows reserve entire network regions (e.g., datacenters) exclusively, just to be cautious about potential impacts; and b) device-level operations on the physical network usually take a long time (e.g., firmware updates).

However, across all workflows and despite their variegated goals, we observe a deep, common pattern in their logic. Almost all workflows (1) query and/or modify network state from "source-of-truth" databases, which maintain a logical view of the network; (2) modify physical network devices based on the logical view; and (3) invoke a library of BBs as necessary (e.g., update firmware, create configuration). Figure 1c further shows the number of BBs in workflows, and Figure 1d illustrates the amount of reuse across BBs.

2.3 Reliability gaps

By analyzing the dataset and discussing with the operation teams, we have identified several classes of issues that must be handled for reliability. As another important observation

that motivates Occam, these reliability concerns are often peripheral to the key management logic.

1. Resolving network state conflicts. Production networks maintain state in "source-of-truth" databases [29, 39], and workflow programs must ensure that they operate on the up-to-date view. Network databases provide query-level transactions, but a management task may require several round trips to the databases. Thus, when queries interleave with each other across tasks, database-level transactions provide no guarantee for task-level isolation. Consider a network migration task that logically deletes a set of devices from the database and later inserts replacement devices. It must be carefully coded so that by default, the intermediate state across database queries is not exposed to other tasks. Otherwise, a traffic engineering task may mistakenly view logically deleted devices as offline and thus trigger disruptive rerouting, even though the physical devices are still serving traffic.

2. Resolving device operation conflicts. Another class of complexity revolves around device-level operations on the network devices. Even if two management tasks use logically consistent network state, uncoordinated operations through the physical devices can also give rise to conflicts (e.g., race conditions that lead to management tasks overwriting each other's updates). For instance, a task that performs firmware upgrades and another one that performs configuration changes cannot concurrently touch overlapping devices. Figure 1e shows that 150-200 pairs of workflow instances perform operations on overlapping devices every day. Figure 1f further shows that the number of devices used by a workflow ranges from a few to tens of thousands. Today, workflow programs must coordinate amongst themselves when performing device-level operations to avoid conflicts.

3. Resolving cross-task conflicts. Tasks without any network state or device-level conflicts may still violate subtle network-wide invariants when they are taken together. For instance, a PoP (point of presence) management task detects link flapping at a PoP-backbone network link and decides to offload the traffic to an alternative link. However, it is unaware that another maintenance task is in progress and will soon bring down the alternative link. The two tasks operate on non-overlapping regions of the network, but their composition would disconnect the entire PoP from the backbone.

4. Failure detection and handling. Another class of reliability concern is failure handling. Runtime exceptions (e.g., device failures and service errors) are common in large networks. The database and device-level operations can fail for various reasons, and it is up to the individual workflow programs to diagnose and handle these failures. Our statistics show that the top-two reasons for workflow program failures are database query errors and failures (63%) and device operation errors (17%); other types of miscellaneous failures account for the rest of the problems (10%). Further, when failures are detected, they are reported to the operator with little assistance in suggesting a potential repair.

2.4 Motivation for Occam

Our key insight is that workflow programs have to simultaneously handle the "key management logic", which corresponds to the central management goals (e.g., draining devices and updating their firmware), but at the same time, they need to defend against a wide range of reliability concerns for correctness (e.g., synchronizing across workflows in ad-hoc manners to prevent simultaneous database and device operations, and continuously monitoring network health to detect and handle runtime failures). Instead of leaving these tasks to individual workflows, without any built-in guarantees, we believe that a principled design should *support reliability at the workflow framework level natively.*

To this end, we design Occam to tackle the workflow reliability problem. Occam exposes a set of unified, narrow programming APIs for operators to program the key management logic. The Occam runtime addresses the reliability requirements with three components: (1) a locking mechanism to ensure consistency; (2) a scheduling algorithm to improve efficiency; and (3) a failure recovery layer to assist operators in handling failed tasks by suggesting concrete rollback steps. Occam management tasks are written in Python, following best practices observed in Meta. Network operators may not be trained as programmers, so a popular and easy-to-use language like Python improves usability. This also allows us to port certain management tasks from Meta's platform to Occam for evaluation. However, the Occam design principles are general and not restricted to the choice of implementation languages. These designs borrow inspirations from various database techniques, but the distinct nature of network management requires new considerations.

3 The Occam Programming Model

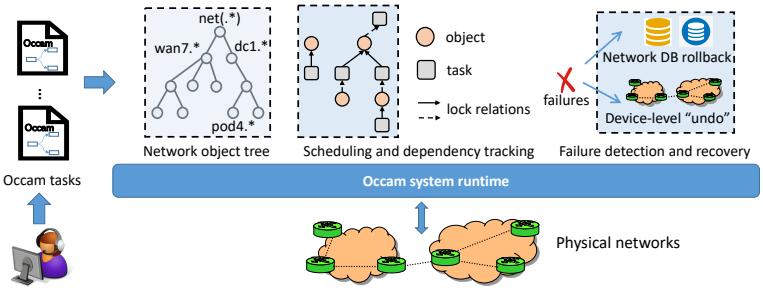
An Occam program creates one or more network objects by scoping the network regions of interest. All subsequent operations either mutate the object state via a narrow API (embedded in Python) or perform stateless computation locally. The stateful API represents the only channel via which Occam tasks exert an influence on the physical or logical network. The Occam runtime operates over this model, which maintains a hierarchy of network objects, enabling analysis of dependencies, locking conditions, task scheduling, and rollback plan suggestion. Figure 2 shows the API and the main components of Occam.

3.1 Data model: Network objects

Occam programs operate on a uniform data model—network objects. An object instance encapsulates all needed network states, and it is the only channel for management programs to modify the network. Thus, the impact of an Occam task is analyzable via its interaction with the object instances. Across tasks, Occam also analyzes how the instantiated network objects are used by the programs, for conflict detection

API	Description
network(regex)	Create and scope a network object
network.get([attr_key])	Get device attribute with key
network.set([attr_key, val])	Set device attribute with key
network.apply(func)	Execute func on physical device
network.close()	Commit state changes

(a) Stateful API on network objects



(b) The Occam runtime system enforces a range of correctness properties

Figure 2. Occam API and an illustrated workflow of the Occam system.

and lock scheduling. Occam programs perform operations on specific network regions as defined by their network objects.

To scope a network region, our design decision is to use regular expressions (regexes) as defined over the network topology identifiers—`network(dc1.*)` captures the region of datacenter `dc1`. This is driven by the observation that datacenter networks are organized in a hierarchical structure (e.g., multi-tier trees), and network devices are usually named and addressed from a well-defined identifier space for manageability. These properties make regexes an ideal fit for scoping the network region, compared to alternative methods such as using a flat dictionary of device names. A regex-based approach fits Meta's network organization and should be generalizable to other network providers. In rare cases where device names are not chosen in a consistent manner, device renaming may be required to use a regex approach. Moreover, we note that a network region as defined by a regex is a symbolic representation: `network(dc1.*)` represents all devices in `dc1`, including existing devices and those that are being added by some ongoing management task. This also enables a powerful locking approach at the region level (§4).

Hence, a network object encapsulates a set of devices that are connected via a set of links, forming a topology. Devices have attributes (e.g., IP addresses, link interface speeds), and links are represented by a class of attributes of their endpoint devices. When creating an object, Occam relies on the user-provided regex to query the network database, and constructs the object state for subsequent management steps. Thus, individual Occam programs always have easy and principled access to the network database without having to handle database-level issues (e.g., SQL query processing). Thus, Occam shields operators from SQL-level concerns, such as directly interfacing with network databases, detecting and resolving database-level errors (recall from §2.3 that they account for 63% errors in the Meta dataset), and implementing task-level transactions that span across SQL queries.

A network object represents the smallest "unit of concern" for network management. It exposes a set of stateful APIs, as defined in Figure 2a, via which Occam programs can exert an influence over the network. At a high level, they can issue read/write requests on the logical network state as queried from the database (e.g., set the `DEV_STATE` attribute to `UNDER_MAINTENANCE`), and they can execute commands on the physical devices (e.g., run `upgrade_data_plane` to change the switch firmware). Scheduling, locking, and fault tolerance guarantees are first enforced at a per-object level, then extended to an entire Occam task, and then across Occam tasks that have dependency relations with each other. Network objects in the same Occam tasks belong to the same transaction, and their operations commit or fail as a whole. Across tasks, dependencies may exist if they operate on overlapping sets of network objects. These reliability concerns, however, are shielded from the programmer; instead, the Occam runtime system arbitrates access to the network objects to achieve reliable network management.

3.2 Programming Occam tasks

An Occam program performs stateless, local computation unless it explicitly invokes the stateful API to make the effect visible to the network, including `get()`, `set()`, and `apply()`. Each stateful operation logically operates on all devices in the network object. For instance, a `set(dev_attr)` call will modify a set of devices and their attributes without programmers having to manipulate each device individually (e.g., issue separate operations for each device and synchronize them). This results in cleaner code and also encourages the programmer to think at clearer steps. In practice, the realization of a stateful operation may be different on each device—e.g., the input argument `dev_attr` is a dictionary type that can be keyed on device identifiers, which sets distinct attribute values (e.g., IP addresses) for each device. For instance, in Meta, management platforms rely on infrastructure services to interact with physical devices. Management code eventually invokes other services that expose API for manipulating each network device in a vendor-specific manner. Occam adopts the same design, where management tasks use a higher-level API but eventually, the operations will be performed on physical devices via vendor-specific services.

The `get()` operation returns a dictionary of device attributes for each device in the network object. As discussed, dictionary keys are the device IDs in the regex identifier

space, and the attributes themselves are dictionary types, too. An attribute contains attribute names (e.g., IP address, device state) and values—both queried from the network database. The `set()` operation modifies device attributes (e.g., assign IP addresses for each device). The `apply()` operation, on the other hand, executes a "device function" `func`, which is a management operation that is executed on the physical network devices. The `func` library is largely fixed and only expands slowly over time, as it contains the basic device-level commands that are usually reused across tasks (e.g., performing firmware upgrades, device SSL key maintenance). In this sense, this library serves a similar purpose as the Building Blocks (BBs) in existing systems [28].

3.3 Occam by examples

To further showcase how operators use Occam, we now include Occam programs for a set of management tasks.

Ex: Device maintenance for a pod. As a concrete example, consider a simplified Occam program that flags specific switches in a pod as under maintenance.

```
1 # device_maintenance.occam
2 dc1pod3 = Network("dc1.pod3.*")
3 dc1pod3.set("DEVICE_STATUS", "UNDER_MAINTENANCE")
4 dc1pod3.apply(f_push)
5 dc1pod3.close()
```

All four steps belong to the same transaction, and Line 5 is the serialization point where the transaction commits. The programmer only needs to understand the conventions for device attributes (e.g., `DEVICE_STATUS`) and gain familiarity with the set of device functions (e.g., `f_push`, `f_drain`); this is the same as with any other network management system. The runtime system maintains a log that tracks the progress of each API operation and whether or not it succeeds. For instance, Lines 2-3 may experience SQL connection failures or incur an exception due to an invalid regex; Line 4 may also fail if the device management interface cannot be reached or if the drain command didn't successfully execute. Upon a failure, Occam assists the operator by suggesting a concrete rollback plan both for the database and device state.

Ex: Dynamic object creation. The next program obtains link status of each device, and dynamically creates a network object (Line 8) that only contains devices whose links are not yet turned up. It then turns up all links for this new object.

```
1  # turnup_links_subnet.occam
2  net = Network(args)
3  link_status = net.get("LINK_STATUS")
4  dev_names = []
5  for l, s in link_status.items():
6    if s != "UP":
7      dev_names += [l.a_end, l.z_end]
8  subnet = Network(to_regex(dev_names))
9  subnet.set("LINK_STATUS", "UP")
10 subnet.apply(f_push)
11 net.close()
12 subnet.close()
```

Ex: Link turn-up. We include the complete Occam code in our code release [42] for link turn-up, which is ported from the original workflow program obtained from Meta. We have further tested in the Meta environment and ensured that the Occam task correctly performs the same functionality.

4 Multi-Granularity Locking

Next, we describe how Occam executes management tasks and ensures consistency through multi-granularity locking.

4.1 Occam tasks

An Occam task is an instance of an Occam program, executed with a specific network input. Each task exists in one of the four states: a) submitted, where the task is enqueued but has not been selected to run; b) active, where the task has already made some progress but has not finished; c) completed, where the the task has successfully finished and committed all its changes; and d) aborted, if the task has encountered runtime failures and cannot succeed. Normally, a task eventually transitions through the first three states, and successfully modifies database and physical device states. Upon task failures, Occam also suggests step-by-step recovery plans to assist the operators (§6).

4.2 Object/Task dependency graph

The key data structure in task tracking is an *object/task dependency graph*, which captures active network objects, active tasks, and how they relate to each other. Figure 3 shows an example. *Object* nodes are organized in a tree structure, which we call a network object tree; we show a concrete example in Figure 3b. This tree maintains a hierarchy of network regions in which management operations are ongoing or planned. Object/object edges are directional and indicate containment relations—e.g., a node `dc1.*` would be the parent of `dc1.pod3.*`. This hierarchical organization facilitates efficient contention detection across network regions and also helps reason about changing the granularity of locks as suited for each network region.

Task nodes are not part of this tree, but they have directional edges pointing from and to certain objects in the tree (Figure 3c). Task/object edges indicate lock requests and acquisitions—an edge from an object to a task indicates that the task currently holds a lock on the object, and the reverse direction indicates that the task is currently waiting for a lock on the object. The edges are further annotated with lock types: `S` for shared locks, `X` for exclusive locks. Tasks do not have direct edges to each other, but they are indirectly dependent upon each other via their locks on the objects. Taken together, the object/task dependency graph represents all active management tasks and their network regions. This dependency graph is gradually built as new tasks arrive.

4.3 Network object tree operations

The network object tree contains all active objects from all tasks, and it upholds two invariants in its construction: a)

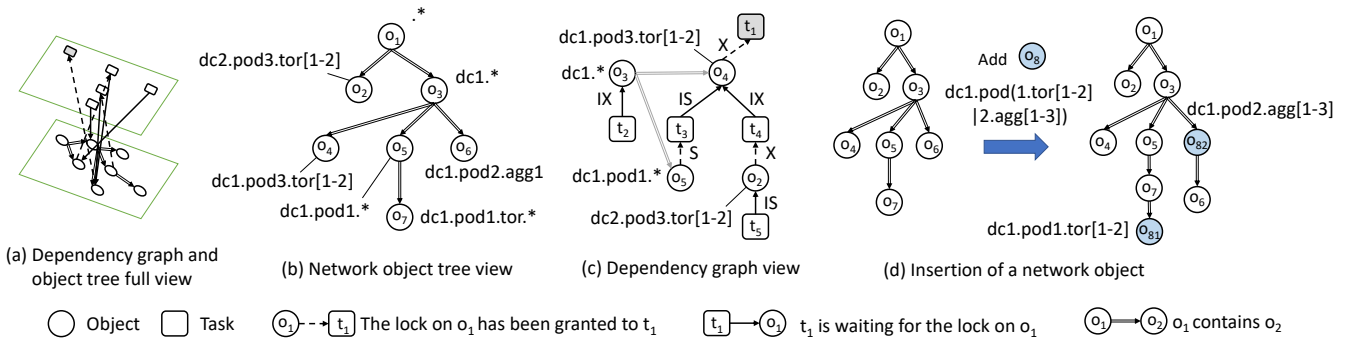


Figure 3. Network object tree and object/task dependency graph.

a parent object strictly contains all its children in terms of their network scope (e.g., `dc1.*` contains `dc1.pod3.*`), and b) siblings of the same parent do not have any overlap (e.g., `dc1.pod3.*` may have a sibling node `dc1.pod2.*`). This design enables Occam to efficiently navigate this tree to identify active network regions. Figure 3b shows a concrete example, and Figure 4 shows the maintenance algorithm, which is further described below for each operation.

Insert. Leveraging the containment relation maintained by this tree, the INSERT algorithm locates a position in the hierarchy to insert a newly created object obj. The algorithm performs a recursive descent from the virtual root `.*`, precisely following a unique path of containment relation, ensuring that obj is always contained by all nodes in this path. This descent stops at a node p when obj is no longer strictly contained by any of p's children.

Split. The split between two overlapping objects o_1 (i.e., the new node obj) and o_2 (i.e., the existing node c) first identifies their intersection o, and then inserts o to the existing subtree rooted at o_2. After the split, the new object is updated to $o_1' = o_1 \setminus o$ (Line 12), and further compared against other children in this hierarchy. The analysis of overlaps and splits is based upon efficient regex operations, which are guaranteed to produce valid regexes as outputs [10]. Figure 3d demonstrates a concrete example.

Delete. The DELETE algorithm is performed in the background, and Occam uses reference counting on each active network object. Every time a task finishes, its objects will be checked by Occam to see whether they are ready for deletion. An object will only be deleted from the tree when the last task that uses it finishes (i.e., commits or aborts). The delete operation itself is simple—removing a node from the tree and grafting its children to the deleted node's parent.

```
1:  function InitObjTree
2:      objtree.root ← regex(".*")
3:      objtree.root.lock ← NULL
4:  function Insert(root, obj) //Entry: root = objtree.root
5:      for c ∈ GetChildren(root) do
6:          if Contains(c, obj) then //Recursive descent
7:              isLeaf(c)?AddChild(c,obj):Insert(c,obj);return
8:          else if Contains(obj, c) then
9:              AddChild(obj, c)
10:         else if Overlapping(obj, c) then
11:             Split(obj, c, root)
12:             obj ← obj \ c
13:     if isNotEmpty(obj) then // remaining portions after splits
14:         AddChild(root, obj)
15: function Split(o1, o2, r) //o1: new obj, o2: existing obj
16:     o ← Intersects(o1, o2)//Regex intersection
17:     Insert(o2, o) //Insert intersection into the existing child
18: function Delete(obj)
19:     parent ← GetParent(obj)
20:     for c ∈ GetChildren(c, obj) do
21:         AddChild(parent, c)
```

Figure 4. The algorithm for maintaining the object tree.

4.4 Multi-granularity locking

Next, we describe how Occam constructs the object/task dependency graph on top of the network object tree. The hierarchical nature of this dependency graph enables a *multi-granularity locking* scheme which, in the database transaction context, is known to provide good performance due to precise locking as needed [21]. More concretely, finer-grained locking (e.g., per data item) enables more parallelism opportunities, as locks are acquired and released as soon as per-item operations finish; but this produces many locks and higher overheads; coarser-grained locking is just the opposite. Ideally, the granularity of locks should be adjustable and chosen to be sufficiently large but not overly so. In Occam, the object-level locking achieves this effect, as the object scope naturally varies based on the network regions—thus, we never lock more or less than what is necessary for each Occam task. An object/task dependency edge could be a *locking* edge, which points from an object to a task, indicating that a task has earned privileges to operate on the object. It could also be an *intentional* edge [21], which points from a task to an object, indicating a lock request that is yet to be fulfilled. Both edges could indicate the mode of access—i.e., `S`/`X` for locking edges, and `IS`/`IX` for intentional edges.

Task arrival. When a new task is submitted, Occam inserts network objects as the task executes, and then adds the task as well as its dependency edges. For an *obj*, Occam adds exactly one dependency edge from *new_task*, which represents an intentional lock. If the Occam program only uses `get()` operations on *obj*, the dependency edge is of the type `IS` (intentional shared). If the program involves `set()` or `apply()`, `IX` (intentional exclusive) will be added. Occam then triggers a Sched operation at the scheduler, which analyzes the object- and task-level dependency across the network to determine whether or not a lock request can be fulfilled (more details in the next section).

Task execution and completion. When a network object has finished its operations, the task has made forward progress by one object, and will insert the next network object if it exists. Eventually, Occam modifies the object/task dependency graph when a task successfully completes its operations on all objects. The algorithm iterates through all objects that are locked by this task, and removes the locking edges to release the locks. If a released object has no further `IX/IS` edges (lock requests) or `S` edges (shared locks), this means the object can be safely deleted from the object tree. However, if the released object still has intentional locks, it is then again up to the scheduler to decide which task it should be granted to, so a Sched() operation is triggered.

5 Contention-Aware Scheduling

We now describe the Sched operation, which improves management efficiency and thus reduces the vulnerable time using a lock scheduling algorithm. At the task level, Occam uses strict 2PL [9], which gradually acquires locks as they become available, but only releases them together when the task commits. The majority of database designs use FIFO order to schedule locks across transactions—i.e., available locks are given to the earliest-arrival task—and recent work has also considered contention-aware scheduling in the LDSF (largest dependency set first) policy [40]. Occam supports both policies, and it adapts LDSF to our setting based on the unique structure of the object/task dependency graph. LDSF uses two insights to facilitate faster overall progress. First, across all objects (i.e., database items), the contention levels vary (i.e., the number of tasks/transactions requesting for and blocked on a particular object), and likewise certain tasks may be blocking more tasks than others (e.g., they may hold some high-contention objects). Second, when an object becomes available, granting an exclusive lock to this object is an expensive operation, as it only allows one task to make progress; granting a shared lock on this object, in contrast, may simultaneously enable multiple tasks to make progress.

Adapting the first insight in LDSF to our setting yields three types of dependency analyses. *(i) Direct dependencies:* A task t_1 depends on another task t_2 if t_1 has an `IS/IX` lock on an object o, but t_2 currently holds an `S/X` lock on the same object. *(ii) Transitive dependencies:* Transitively, if t_1 reaches t_2 over a sequence of object/task dependencies (e.g., t_1 waits for some object held by t, which in turn waits for t_2's objects), this reachability, denoted as $t_1 \rightsquigarrow t_2$, also represents dependency across tasks. *(iii) Containment relations:* Last but not least, Occam analyzes dependencies that arise due to the hierarchical nature of network objects. A task that holds an `S/X` lock on some object o also blocks all tasks that have an `IS/IX` lock on o', if they have containment relations with each other, either $o \subset o'$ or $o' \subset o$. Lines 37-43 in Figure 5 describe how we compute the dependency set for a task. As

```
1:  function Sched(objtree, task_list)
2:      for obj ∈ objtree, obj is unlocked do
3:          // Get runnable read/write tasks
4:          wait_wt_tasks, wait_rd_tasks ← GetWaitTask(obj)
5:          if objtree.policy == FIFO then
6:              sched_task ← FIFO(wait_wt_tasks, wait_rd_tasks)
7:          else if objtree.policy == LDSF then
8:              for t ∈ task_list do // Update task dependency set
9:                  t.depset ← FindDepSet(t)
10:             sched_task ← LDSF(wait_wt_tasks, wait_rd_tasks)
11:         if sched_task.type == read then
12:             GrantSLock(wait_rd_tasks) // Run all read tasks
13:         else
14:             GrantXLock(sched_task)
15: function GetWaitTask(obj)
16:     wait_wt_tasks ← ∅; wait_rd_tasks ← ∅
17:     for o ∈ Containment(obj) do
18:         for rd_task ∈ ISLockSet(o), rd_task is ready to run do
19:             wait_rd_tasks ← wait_rd_tasks ∪ {rd_task}
20:         for wt_task ∈ IXLockSet(o), wt_task is ready to run do
21:             wait_wt_tasks ← wait_wt_tasks ∪ {wt_task}
22:     return wait_wt_tasks, wait_rd_tasks
23: function FIFO(wait_wt_tasks, wait_rd_tasks)
24:     wait_tasks = wait_wt_tasks ∪ wait_rd_tasks
25:     //Choose the one with earliest arrival time
26:     earliest_task ← argmin_{t∈wait_tasks} t.arr_time
27:     return earliest_task
28: function LDSF(wait_wt_tasks, wait_rd_tasks)
29:     //Use a fake task to aggregate depset for all read tasks
30:     urd_task ← NewTask( ); urd_task.depset ← ∅
31:     for rd_task ∈ wait_rd_tasks do
32:         urd_task.depset← urd_task.depset ∪ rd_task.depset
33:     //Choose the one with largest depset
34:     wait_tasks = wait_wt_tasks ∪ {urd_task}
35:     largest_task ← argmax_{t∈wait_tasks} |t.depset|
36:     return largest_task
37: function FindDepSet(task)
38:     result ← {task}
39:     for obj ∈ task.granted_objs do
40:         for o ∈ Containment(obj) do
41:             for t ∈ IXLockSet(o) ∪ ISLockSet(o) do
42:                 result ← result ∪ FindDepSet(t)
43:     return result
```

Figure 5. The algorithm for hierarchical lock scheduling.

	Normal Pattern		Rollback plan		Failure Pattern		Rollback plan
(P1)	seq	::= seq step | step	r(step)→r(seq) r(step)	(P6)	b_seq	::= seq b_step | b_step	r(b_step)→r(seq) r(b_step)
(P2)	step	::= cfg_change | offline | testing	r(cf_change) r(offline) r(testing)	(P7)	b_step	::= b_cfg_change | b_offline | b_testing	r(b_cfg_change) r(b_offline) r(b_testing)
(P3)	cfg_change db_list	::= db_list PUSH_CFG ::= db_list DB_CHANGE | DB_CHANGE	r(db_list)→PUSH_CFG r(DB_CHANGE)→r(db_list) r(DB_CHANGE)	(P8)	b_cfg_change	::= db_list	r(db_list)
(P3)	offline	::= DRAIN seq UNDRAIN	DRAIN→r(seq)→UNDRAIN	(P9)	b_offline	::= DRAIN seq | DRAIN b_seq | DRAIN	r(seq)→UNDRAIN r(b_seq)→UNDRAIN UNDRAIN
(P5)	testing test_list	::= PREPARE test_list UNPREPARE ::= test_list TEST | TEST	– – –	(P10)	b_testing	::= PREPARE test_list | PREPARE	UNPREPARE UNPREPARE

Table 1. Log matching patterns and their reverse functions.

concrete examples in Figure 3c, (t_3, t_1), (t_4, t_1), (t_5, t_4) are direct dependencies, (t_5, t_1) is a transitive dependency, and (t_2, t_1), (t_2, t_3) are containment dependencies.

When picking a task to run, the scheduler not only considers the tasks waiting for the current object, but also those waiting for objects with containment relations with this object. Lines 15-22 in Figure 5 describes the method to find all such tasks. The GetWaitTask algorithm returns separate lists for write and read tasks respectively, which will be the input for the scheduling algorithm.

Regardless of the policy, at the end of each Sched invocation, Occam will grant S locks for all waiting read tasks if the sched_task is read; otherwise, it only grants one X lock for sched_task. The lock granting process simply flips the direction of the edge in the object/task dependency graph—i.e., from IS/IX edges to S/X edges. Once a task has obtained a network object that it is waiting for, it can proceed to issue Occam operations on this object. Assuming that no failures occur, then eventually all tasks successfully commit with their modifications. It is known that in 2PL, deadlocks could happen. Occam handles deadlocks by detecting and breaking them via aborting and re-executing the task that causes a deadlock. Occam also supports accommodating urgent tasks like outage recovery by prioritizing their scheduling.

6 Assisting Failure Recovery

Upon failures, Occam suggests a sequence of undo operations to the operator to roll back the effect of the management task. As discussed in §2, failures are high-priority incidents, so human operators must be involved to identify or roll out repair plans. Nevertheless, Occam can assist this process by suggesting a potential recovery plan.

In the simplest case, a sequence of steps $s_1, \cdots, s_k$ is undone by following the reverse order to undo each of the steps $s_k^{-1}, \cdots, s_1^{-1}$—assuming that each step s is reversed by some operation s^{-1} manually or with pre-specified configlets [27]. This is shown as pattern P1 in Table 1, which is a stateless sequence of operations that can be rolled back with a linear reversal. However, the reversal order is not always linear, and an example pattern is shown in P3 in Table 1. Consider a sequence of database operations (i.e., `set`/`get`) followed by a device function that pushes the computed configurations (i.e., `apply`). Suppose that all these operations succeed, but the next one after them fails, Occam must recover the database state first and only then update the device configurations to a clean state. In other words, in this case, the reversal order and the execution order are the same.

From the dataset, we found that a correct order of reversal depends on the semantics of the management operations. We have therefore devised a solution that works by a pattern matching process. Table 1 shows the pattern matching rules that process an execution log and suggests a repair plan. This algorithm associates a "type" with a subset of special device functions (i.e., `apply(func)`) to delineate pattern boundaries: (a) `PUSH_CFG` labels device functions that push configurations to physical devices (e.g., `f_push`); (b) `DRAIN` and `UNDRAIN` label functions that bring devices offline or online; (c) `PREPARE` and `UNPREPARE` denote functions that set up and tear down temporary test environments (e.g., `alloc_test_ip`, `dealloc _test_ip`); and (d) `TEST` denotes functions that perform a specific test (e.g., `run_ping`). The `DB_CHANGE` denotes `set` calls instead of `apply(func)` calls.

With these types, an execution log comprises a sequence of operation steps (P1). Each such step is either a configuration change to an active device, an offline maintenance operation, or a testing operation (P2). A configuration change further involves a series of database updates (P3). Offline maintenance requires the entire device or some interfaces to be drained before and undrained after the maintenance procedure (P4), and it may include a series of operations of type P1. Testing can include one or more rounds (e.g., physical-layer to IP-layer tests). Before these tests, temporary environments are set up (e.g., a temporary IP address is allocated to the device to ensure it is accessible), and they are torn down after testing (P5). When a sequence of operations fail in the middle, Occam will match the operations against a set of rules in Table 1 and identify broken sequences that cause pattern mismatches, as shown in P6-P10 in Table 1. Based on these rules, Occam will generate a "syntax tree"-like data structure for the logged execution. Occam suggests a repair by walking the tree, based on the reversal rules for each mismatch, to generate a concrete recovery plan. In essence, this plan

Label	Example Occam APIs
DB_CHANGE	set(·)
PUSH_CONFIG	apply(f_push)
DRAIN	apply(f_drain)
UNDRAIN	apply(f_undrain)
PREPARE	apply(f_alloc_ip)
UNPREPARE	apply(f_dealloc_ip)
TEST	apply(f_ping_test), apply(f_optic_test)

Table 2. Operations used in a firmware upgrade task, with each operation annotated with types for pattern matching.

will undo broken subsequences first, and then recursively fix their enclosing sequences.

We now consider a concrete example of performing a firmware upgrade, where the relevant device functions as well as their types are shown in Table 2. This task drains traffic from selected devices (`f_drain`), changes the firmware based on the database state (e.g., firmware version and its binary location) (`set`), pushes device configuration (`f_push`), prepares testing environment such as temporary IP addresses (`f_alloc_ip`), executes tests such as ping to check connectivity and fiber optic testing (`f_ping_test`, `f_optic_test`), releases testing environment (`f_dealloc_ip`), and finally undrains the traffic back (`f_undrain`) to the selected devices.

Now let us consider a scenario where `f_optic_test` has failed—the task execution stops at: `f_drain` → `set` → `set` → `f_push` → `f_alloc_ip` → `f_ping_test` → `f_optic_test` → `X`, where the last step `X` denotes that `f_optic_test` has failed. To generate the rollback plan, Occam labels the sequence using types shown in Table 2, obtaining `DRAIN` → `DB_CHANGE` → `DB_CHANGE` → `PUSH_CONFIG` → `PREPARE` → `TEST` → `TEST`. Then, it constructs the "syntax tree", as shown in Figure 6, by recursively matching the sequence into the failure pattern, using the rules in Table 1. This process is akin to how a compiler parses an input program into its syntax tree. Concretely, the failure pattern `b_seq` (i.e., the root of the tree in Figure 6) matches the second case of P6, `b_step`, thus we get the rollback plan `r(b_step)`. Following that, `b_step` matches the second case of P7, `b_offline`, producing `r(b_offline)`. Next, `b_offline` matches the first case of P9, `DRAIN seq`, and therefore its rollback plan is `r(seq)` → `UNDRAIN`. The recursive construction will continue on `r(seq)`. Eventually, we will construct a complete rollback plan: `UNPREPARE` → `r(DB_CHANGE)` → `r(DB_CHANGE)` → `PUSH_CFG` → `UNDRAIN`.

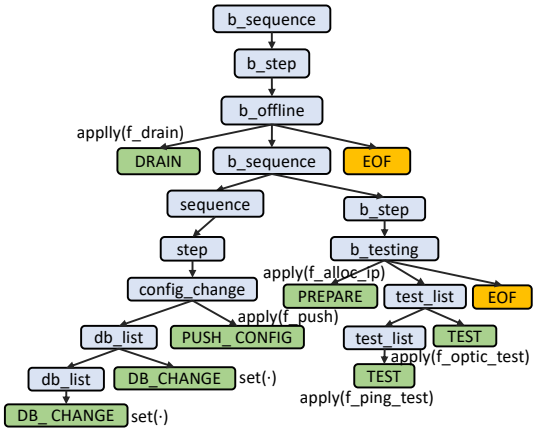


Figure 6. The "syntax tree" of the firmware upgrade failure.

7 Implementation

The Occam emulation and simulation platform is closely modeled after that in Meta (Figure 7). It consists of 7100 lines of code and has been released in open source [42]. It can configure network topologies of various sizes in Mininet [14] with software switches [32]. Occam tasks invoke the stateful API exposed by the frontend, which eventually communicates with the network databases and devices via RPC. For large-scale experiments, Occam also provides an event-based simulator that can create representative Occam workloads based on user-configured task arrival rates and network scopes. The Occam runtime also caches frequently-used regexes and their translated automata.

The key implementation challenge is to identify a set of key components in the Meta network, and then construct an experimental platform that closely emulates this production setting. To this end, we have engaged with engineers at Meta to understand their network management architecture. We then emulate the system using open-source software components. Prior to Occam, we are not aware of a publicly available system modeled after realistic production setups for network management. Thus, we believe that our system could serve a playground for researchers to conduct future network management experimentation in the future.

7.1 Implementation details

Emulation. At the heart of this platform is the Occam management plane, which interacts with the control and data planes of individual devices. Management tasks are programmed against the Occam API to interact with the network (shown as "frontend" in Figure 7). The Occam backend is the runtime system, which is in charge of scheduling and locking. The management plane services are also part of the runtime system, which eventually executes the stateful API by querying the source-of-truth databases and interacting with individual switches for device-level command and configuration. In our platform, control and data plane emulation builds upon the open-source Mininet codebase [14], which can support different topologies and different software switch models. Our setup uses software `bmv2` [32] programmable switches for individual devices. Control plane emulation relies on the P4Runtime interface [33], which enables the individual switches to communicate with the centralized Occam platform via RPC. The Occam tasks invoke the stateful API via the runtime system, which eventually reaches

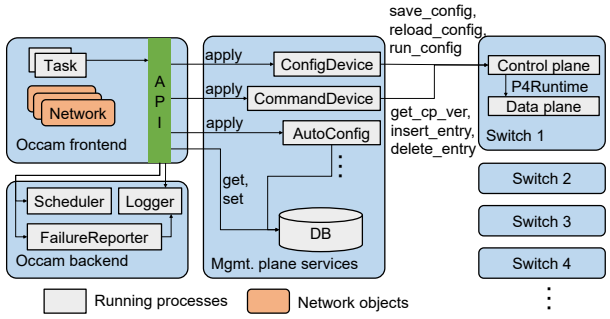


Figure 7. Architecture of the Occam platform.

the devices to execute the various commands. Occam also contains utility functions to instantiate Fat-trees of various sizes and to configure the topology with ECMP forwarding.

Simulation. For large-scale experiments, Occam also provides an event-based simulator that can create representative Occam workloads based on user-configured task arrival rates and network scopes. The default workload parameters are based upon the Meta trace, but new distributions can be easily added. The simulator can be configured with three lock granularities: DC locks, device locks, and network object locks. For each level of granularity, it further supports two scheduling policies: FIFO and LDSF. Thus, in total, the simulator supports six different schedulers. Occam leverages an open-source library [34] for regex operations. Regexes are eventually translated into finite state machines (FSMs) for intersection and other regex operations. Occam also caches recently seen regexes and their FSMs. The Occam simulator only simulates the runtime scheduling algorithms but it does not interact with emulated network devices, for more scalable experimentation.

8 Evaluation

We now evaluate Occam using simulation, emulation, as well as a set of case studies to demonstrate its effectiveness.

8.1 Simulation at scale

We start by a set of simulation to understand how Occam improves network management at scale. We created a network topology with 16 datacenters, each with 96 pods and 92 switches, which is at a scale that matches the Meta dataset.

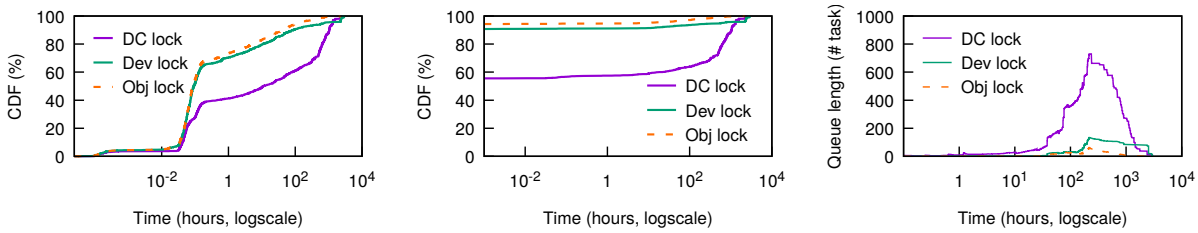
Synthesized workloads and baseline. To synthesize the simulation workloads, we have obtained a longer trace from Meta (5 months). This trace contains detailed data about the distribution of task execution times, task arrival times, as well as the network regions that the management tasks operate on. We then synthesize each simulation experiment by randomly sampling these distributions to derive an appropriate set of management task parameters. For each simulation run, we synthesize 2000 management tasks with different arrival times, execution times, target network scopes (regexes), and read/write ratio, for evaluation. To the best of our knowledge, there is no open-sourced network management tool that can fulfill our experimental purpose. Therefore, we compare our design against a naïve version of Occam, where per-device and per-datacenter (DC) locks are used together with the scheduling algorithm.

Task completion times. Figure 8 measures the task completion times under different Occam scheduling granularities. For this experiment, Occam uses LDSF for lock scheduling and grants available locks to the task with the largest dependent set. A task starts execution when it has obtained all necessary locks. As Figure 8a shows, datacenter-level locking severely constrains concurrency, as at most one X lock can be held per datacenter. Device-level locking significantly improves this and reduces the average task completion time from 312 hours to 129 hours. The multi-granularity, object-level locking strategy performs the best, further reducing the average task completion time to 31 hours (24% of that with device-level locks). We found that this is because when locks are granted to a task for each device individually, it misses the bigger picture at the task level. Consider a task that holds a set of device locks L_1 and waits for L_2. This unnecessarily blocks other tasks that wait for locks in L_1. Object-level locks avoid this problem most of the time—in the absence of splits, it always grants a batch of locks as exactly needed by a task.

Task waiting times. Figures 8b-c zoom in on the task waiting times and queue lengths, respectively. In each case, datacenter-level locks are outperformed by device-level locks, which in turn are dominated by network object-level locks. The difference is drastic: the 90th percentile waiting times for datacenter-level locks is 1037 hours, while object-level and device-level do not experience any waiting time for more than 91% and 94% management tasks, respectively. We also show how queue lengths increase much slower in object-level locking, at a peak of 62 waiting tasks; per-device and per-DC locking strategies reach 134 and 730 tasks, respectively. In each case, the queue length drops when no additional tasks arrive until the queues clear.

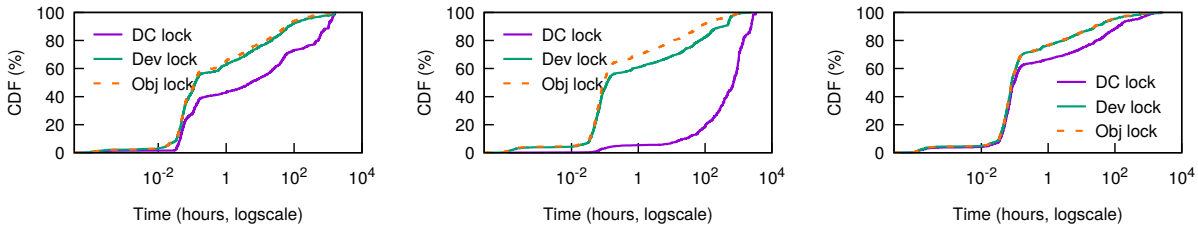
More workloads. We stress test Occam by scaling the task arrival rate by 2, 4, and 6 times, respectively. Figure 9a shows the task completion times with scale factor of four. Scale factors two and six show similar trends, so figures are omitted. Network object-level locking reduces the average completion time by 4.7-7.1× compared to datacenter-level locks, and by 1.7-4.0× compared to device-level locks. We further test write-heavy and read-heavy workloads (each set to be around 95%), respectively, and show the results in Figure 9b and 9c. In both cases, datacenter-level locks are outperformed by the other two strategies. With more reads (thus fewer task-level conflicts), device-level and object-level locks perform similarly to each other. Overall, the task completion times are smaller for read-heavy workloads as they produce fewer conflicts.

Scheduling overheads. Using the Meta trace, we have also measured the scheduling overheads (the time it takes to run the Sched function) for different lock granularities. As



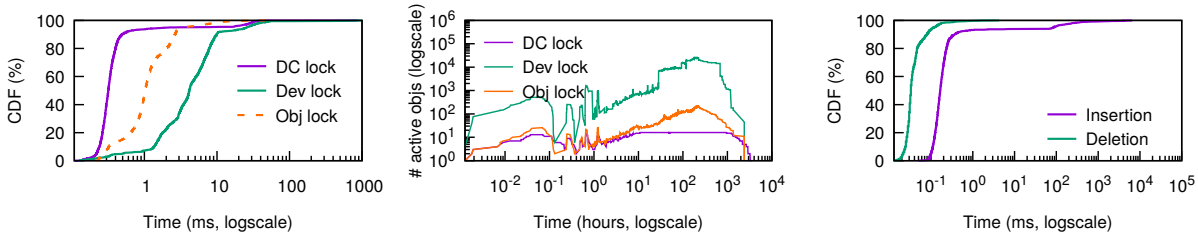
(a) Task completion time **(b)** Task waiting time **(c)** Task queue length

Figure 8. The scheduling effectiveness of Occam for the Meta dataset (when LDSF is used). The multi-granularity locking strategy (Obj lock) outperforms fixed-granularities, either at a per-device (Dev lock) or per-datacenter (DC lock) level.



(a) Arrival rate (scale factor: 4) **(b)** Write heavy (write ~95%) **(c)** Read heavy (read ~95%)

Figure 9. The scheduling effectiveness of Occam for more different workloads (when LDSF is used): (a) scales the Meta trace in terms of the arrival rate by four times; (b) write heavy workloads; (c) read heavy workloads.

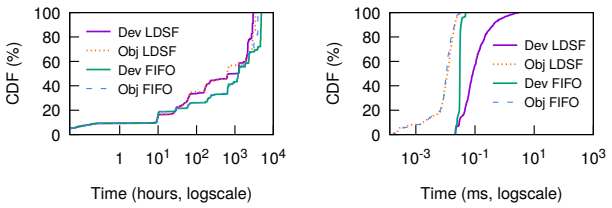


(a) Scheduling overhead **(b)** Active network objects **(c)** Tree maintenance overhead

Figure 10. The scheduling overhead of Occam with different lock granularities (when LDSF is used). (a) shows scheduling times, where scheduling with fewer locks is faster; (b) shows the number of active scheduling objects over time; scheduling objects are either datacenters, devices, or network objects; X-axis is the scheduling steps, one invocation per step; (c) is a breakdown for the network object tree maintenance cost for insertions and deletions.

Figure 10a shows, having fewer locks results in lower overheads: datacenter-level locking is the fastest, device-level locking is the slowest, and network object-level locking falls in between. Across all granularities, the scheduling overhead is low and decisions are computed under 100ms. Figure 10b further plots how the number of scheduling units (i.e., devices, datacenters, or objects) grows over time, with similar trends as above. We can see that device-level locking produces 1-2 orders of magnitude more objects. Figure 10c shows the time it takes to maintain the network object tree, measuring the insertion and deletion time for an object. Insertion takes a longer time as it needs to perform regex comparison.

Scheduling policies. In the above experiments, we found that FIFO and LDSF scheduling policies perform similarly with each other and therefore FIFO lines were omitted. We also found that with skewed contention regions, LDSF outperforms FIFO. Figure 11a shows the waiting times obtained on a synthetic trace with skewed contention, with two observations. First, LDSF can prioritize contention regions and this leads to better performance. Second, device- and object-level locks perform similarly because the containment relations are fewer (due to the skewed contention regions) and the object-level analysis does not add significant benefits. Figure 11b shows the scheduling overheads for each policy. We found that FIFO and LDSF have similar performance when locking at the object level because the number of scheduling objects is sufficiently small. With device-level locks, which produce more scheduling objects, LDSF performs slower as its policy is more complex.



(a) Waiting time **(b)** Schedule overhead

Figure 11. LDSF outperforms FIFO when tasks have skewed dependent set. Network object locking is faster than device-level locking, which requires a large amount of locks.

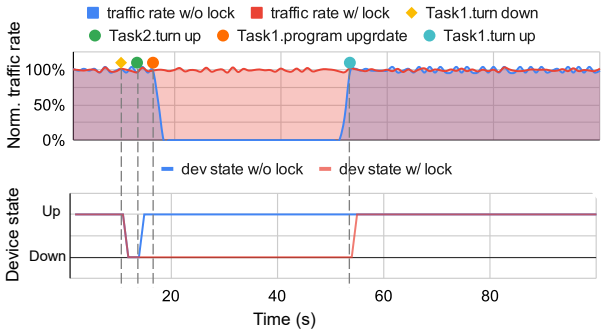


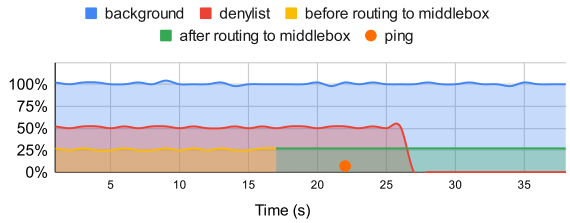
Figure 12. Traffic rates and device states during upgrade_data_plane (Task 1), and turn_up_links (Task 2).

8.2 Emulation case studies

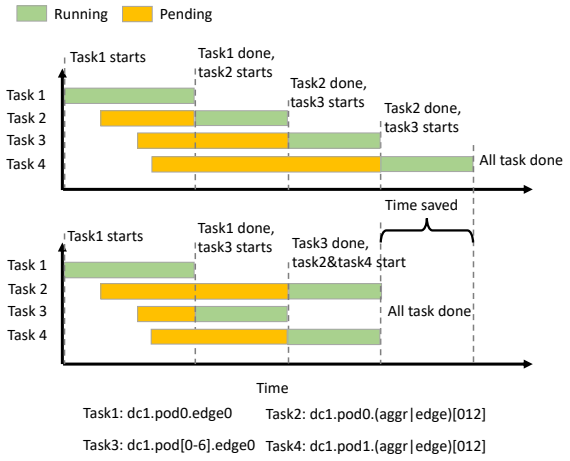
Next, we evaluate Occam with concrete use cases. For our emulation, we created a k=6 Fat-tree datacenter with 18 ToR switches, 18 aggregation switches, and 9 core switches. Each switch is running a P4 program that performs ECMP routing, and end hosts communicate with each other via UDP. We also verify Occam's ability to assist in generating rollback plans with a specific failure example.

Case study #1: Conflicts between switch program upgrade and link turnup. Both are typical tasks in the Meta dataset. The *upgrade_data_plane* task drains traffic from a target switch, upgrades the switch data plane program, and finally undrains the traffic. The *turn_up_link* task attempts to activate a link for the same switch. We measure traffic rates at this switch during the management tasks, and run experiments with and without locking. As Figure 12 shows, in the absence of locks, the *turn_up_links* program restores the traffic back after the *upgrade_data_plane* task drains the traffic. Thus, during the program upgrade, user traffic is dropped by the switch, causing service disruption. In contrast, with Occam's locking mechanism, the two tasks are isolated from each other and correctly executed.

Case study #2: Concurrent tasks with FIFO/LDSF scheduling. This emulation zooms in on the differences between FIFO and LDSF scheduling with four management tasks. A *middlebox_rerouting* task (task 1) that reroutes a certain type of traffic to middleboxes, two *ping_test* tasks



(a) Traffic rate (normalized to background traffic) while running tasks (Only show LDSF as FIFO has similar results)



(b) Scheduling timeline

Figure 13. Traffic rate and scheduling timeline of FIFO and LDSF. (a) shows that there is no traffic disruption when running the management tasks. (b) explains why LDSF can save waiting time when multiple tasks with different dependent set waiting for the same object.

to check connectivity (tasks 2 and 4), and a *denylist* task that blocks suspicious traffic (task 3). Tasks 1-4 arrive in order and they operate on overlapping network regions. We perform the emulation twice, with FIFO and LDSF policies, respectively. Figure 13a shows the traffic rate at the overlapped switches. As we can see, the background traffic rate remains stable; the traffic rate for the blocked flows goes to zero; rerouted traffic is also stable before and after it is rerouted to the middlebox. These measurements show that Occam have handled the conflicts correctly. Figure 13b further plots the concrete scheduling events. From this timeline, we can see the different decisions computed by the schedulers, when two tasks 2 and 3 were both blocked by task 1 due to the contention on acquiring a lock at the same device. The FIFO policy chose task 2 because it requested the lock earlier than task 3; on the other hand, the LDSF policy chose task 3, because task 2 did not block any other tasks (i.e., its dependency set is one), but task 3 blocked task 4 (i.e., its dependency set size is two). This is a concrete example of the contention-aware analysis as afforded by LDSF. This in turn shortens the total waiting time of concurrent management tasks under LDSF.

Case study #3: Network regions of different sizes. In this case study, we emulate management tasks that operate on network regions of different sizes, e.g., whole pods, all aggregation switches, and whole DC. We created multiple such tasks in our emulator, and validated their correctness by examining their functional effectiveness and monitoring the absence of traffic disruption. We omit the timeseries as the patterns are similas as in case studies #1-#2.

Rollback plan generation. We use the same firmware upgrade task discussed in §6 as an example. The task drains traffic from selected devices, changes the firmware based on the database state (e.g., firmware version and firmware binary location), pushes device configuration, prepares testing environment such as temporary IP addresses, executes tests such as ping to check connectivity, releases testing environment, and finally undrains the traffic back to the selected devices. We have injected failures at each step of the execution and examined the rollback plan suggested by the Occam system. For each failure point, Occam is able to use the pattern-matching algorithm to suggest a series of concrete steps to undo the effects. We have manually verified that following these steps will result in the correct rollback of the database and device state.

8.3 Meta case studies

Next, we port three Meta programs to Occam and deploy it to the Meta datacenter. The original Meta programs invoke the existing management services and APIs provided by the Meta workflow system. The ported programs, on the other hand, rely on the Occam APIs. Both operate on real datacenters and services running in Meta. We compare the lines of code needed to perform stateful operations on the network to complete the tasks in the existing and Meta platforms.

Case study #4: It sets up test IP addresses on devices, conduct connectivity tests for these devices, and then deallocates the addressed upon successful tests. Interleaving of the same workflow on the same link can cause one workflow's allocated IP addresses to be deallocated by the other. From our interview with network operators, this has been a cause of several tickets in production. The original program requires 131 LoC to invoke stateful service to modify the network and devices, and Occam reduces it to 6.

Case study #5: It queries the Meta database to determine device health, change link states to active, generate configurations, and then monitor the results. This is a representative and frequently used workflow managing backbone devices. The original program has 307 lines of service invocation code and Occam reduces this to 11.

Case study #6: It changes device states, creates configurations for the affected devices, and then deploys the configurations. This is a representative and frequently used workflow managing datacenter devices. Occam reduces the lines of service invocation code from 311 to 6.

9 Related Work

Automated network management. Network management has received a lot of attention over the years [2, 7, 11, 13, 17, 22, 24, 26, 29, 38, 39, 41], and recent systems as exemplified by AT&T's CORNET [28], Google's ZTN [25], and Alibaba's NetCraft [27] have made significant progress in automation using workflow systems. Occam is inspired by workflow automation [28], but develops a programming system to shield operators from reliability concerns, which are automatically handled in its runtime. Researchers have developed programming abstractions for the network control and data planes [8, 23, 35–37], but Occam targets the management plane, which has received much less attention [1].

Datacenter management. Datacenter infrastructures and tasks are provisioned and managed using domain-specific tools. For instance, tools like Terraform [6], Pulumi [5], and OpenTofu [4] programmatically define the infrastructure in a domain-specific configurations, and orchestration tools like Kubernetes [3] provide support for task scheduling and execution. Compared to these tools, Occam is a system that is tailored for network management tasks.

Programming systems. Using a programming system to shield peripheral concerns is inspired by systems like MapReduce [15] for data processing tasks, Ray [31] for AI/ML workloads, Sapphire [43] for mobile/cloud distributed applications, and dSpace [19] for smart buildings. By shifting low-level issues to the framework level, programmers can focus on developing the key business logic instead. In Occam, the network objects separate stateful and stateless computation, and have resemblance to actor-based systems [31].

Transactions and recovery. Occam borrows database techniques for transaction processing, including multi granularity locking [21], actor-based databases [16], and write-ahead logging [30]. It adapts these techniques to the context of network management for transactional semantics. Rollback recovery has been studied in this setting [30], but management tasks are more complex than database operations.

10 Conclusion

In this paper, we have presented Occam, a workflow system aiming at achieving reliable network management. Occam provides a programming system that shields programmers from reliability concerns, and its runtime handles locking and scheduling, and assists failure recovery systematically. Our evaluation in simulation, emulation, and real-world case studies shows that Occam is able to schedule management tasks effectively and efficiently and that the reliability concerns are correctly handled by the framework.

Acknowledgments: We thank our shepherd, Richard Mortier, and the anonymous reviewers for their insightful feedback. This work was supported by NSF grants CNS-2420309, CNS-2345339, CNS-2214272, CNS-2106338, CCRI-2016727, a Google PhD Fellowship, and a VMware Early Career Faculty grant.

References

[1] Aditya Akella and Ratul Mahajan. 2014. A Call to Arms for Management Plane Analytics. In *Proc. HotNets*.

[2] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. 2019. Risk based planning of network changes in evolving data centers. In *Proc. SOSP*.

[3] The Kubernetes Authors. 2024. [Kubernetes] Production-Grade Container Orchestration. (2024). https://kubernetes.io/.

[4] The OpenTofu Authors. 2024. OpenTofu: The open source infrastructure as code tool. (2024). https://opentofu.org/.

[5] The Pulumi Authors. 2024. Pulumi: Infrastructure as code in any programming language. (2024). https://www.pulumi.com/.

[6] The Terraform Authors. 2024. Terraform by Hashicorp. (2024). https://www.terraform.io/.

[7] Hitesh Ballani and Paul Francis. 2007. Conman: a step towards network manageability. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007), 205–216.

[8] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. SIGCOMM*.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[10] Cezar Câmpeanua and Nicolae Santeanb. 2009. On the intersection of regex languages with regular languages. *Theoretical Computer Science* (2009).

[11] Xu Chen, Yun Mao, Z. Morley Mao, and Kobus van der Merwe. 2010. Declarative Configuration Management for Complex and Dynamic Networks. In *Proc. CoNEXT*.

[12] Xu Chen, Z. Morley Mao, and Jacobus Van der Merwe. 2009. PACMAN: a Platform for Automated and Controlled network operations and configuration MANagement. In *Proc. CoNEXT*.

[13] Rithvik Chuppala, Silvery Fu, and Sylvia Ratnasamy. 2023. DBNet: Leveraging DBMS for Network Automation. *arXiv preprint arXiv:2308.15780* (2023).

[14] Mininet Project Contributors. 2023. Mininet. (2023). http://mininet.org/.

[15] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In *Proc. OSDI*.

[16] Tamer Eldeeb and Phil Bernstein. 2016. Transactions for Distributed Actors in the Cloud. *Technical Report: MSR-TR-2016-1001* (2016).

[17] William Enck, Thomas Moyer, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Yu-Wei Eric Sung, Sanjay Rao, and William Aiello. 2007. Configuration management at massive scale: System design and experience. In *Proc. USENIX ATC*.

[18] The Apache Software Foundation. 2023. Apache Airflow. (2023). https://airflow.apache.org/.

[19] Silvery Fu and Sylvia Ratnasamy. 2021. dSpace: Composable Abstractions for Smart Spaces. In *Proc. SOSP*.

[20] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. In *Proc. SIGCOMM*.

[21] J. Gray, R. Lorie, and G. Putzolu. 1975. Granularity of Locks in a Shared Data Base. In *Proc. VLDB*.

[22] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-Driven WAN. In *Proc. SIGCOMM*.

[23] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *Proc. NSDI*.

[24] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 539–550.

[25] Bikash Koley. 2016. The zero touch network. (2016).

[26] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. zUpdate: Updating data center networks with zero loss. In *Proc. SIGCOMM*.

[27] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic Life Cycle Management of Network Configurations. In *Proc. SIGCOMM SelfDN Workshop*.

[28] Ajay Mahimkar, Carlos Eduardo de Andrade, Rakesh Sinha, and Giritharan Rana. 2021. A composition framework for change management. In *Proc. SIGCOMM*.

[29] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. 2020. Experiences with modeling network topologies at multiple levels of abstraction. In *Proc. NSDI*.

[30] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems* 17, 1 (1992).

[31] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *Proc. OSDI*.

[32] p4lang. 2023. P4 behavioral model. (2023). https://github.com/p4lang/behavioral-model.

[33] p4lang. 2023. P4Runtime. (2023). https://p4.org/p4-runtime/.

[34] qntm. 2023. FSM/Regex conversion library. (2023). https://github.com/qntm/greenery.

[35] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. In *Proc. SIGCOMM*.

[36] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. 2011. Consistent Updates for Software-Defined Networks: Change You Can Believe In!. In *Proc. HotNets*.

[37] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A language for provisioning network resources. In *Proc. CoNEXT*.

[38] Peng Sun, Ahsan Arefin, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, and Ming Zhang. 2014. A network-state management service. In *Proc. SIGCOMM*.

[39] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. 2016. Robotron: Top-down network management at facebook scale. In *Proc. SIGCOMM*.

[40] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-aware lock scheduling for transactional databases. In *Proc. VLDB*.

[41] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *Proc. SIGCOMM*.

[42] Jiarong Xing, Kuo-Feng Hsu, Yiting Xia, Yan Cai, Yanping Li, Ying Zhang, and Ang Chen. 2024. Occam open-source release. (2024). https://github.com/jiarong0907/Occam.

[43] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. 2014. Customizable and Extensible Deployment for Mobile/Cloud Applications. In *Proc. OSDI*.