Lightweight Automated Reasoning for Network Architectures

Rahul Bothra
UIUC

Venkat Arun
UT Austin

Brighten Godfrey
UIUC and Broadcom

Akshay Narayan
Brown University

Ahmed Saeed
Georgia Tech

Abstract

Architecting a modern data center network is increasingly complicated. Seeking the highest performance and support for emerging workloads, network architects planning a build-out must choose from a large selection of switching components, NICs, network stacks, congestion control algorithms, routing schemes, measurement systems, virtualization software, centralized bandwidth allocators and security mechanisms, all from various vendors. Today, manual planning by human experts is time-consuming at best, and can easily result in overlooked design choices or missed complex inter-dependencies.

We propose a radical departure from typical whiteboard-and-spreadsheet planning, and ask: is it possible to reason automatically about possible network architectural designs? Such an approach is nontrivial since formal reasoning about even a single component (like routing systems) is difficult, and we seek to understand how a variety of functional components fit together. We explore the challenge through examples and propose an automated lightweight reasoning framework that models architectural complexities at a broad, but shallow, level of abstraction. Such a framework could serve as a useful design tool for network architects, for careful cross-team planning, and even to help vendors plan product features and requirements.

CCS Concepts

• **Networks** → **Network architectures**; **Network management**.

HotNets '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696865

Keywords

Network Verification, Network Management

ACM Reference Format:
Rahul Bothra, Venkat Arun, Brighten Godfrey, Akshay Narayan, and Ahmed Saeed. 2024. Lightweight Automated Reasoning for Network Architectures. In *The 23rd ACM Workshop on Hot Topics in Networks (HotNets '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3696348.3696865

1 Introduction

The modern network architect faces a daunting task. They must decide what combination of systems and hardware (we refer to this combination as the *network architecture*) to deploy to handle tasks such as bandwidth allocation, traffic engineering, network virtualization, security, end-host network processing, and more. This choice may depend on the applications the architect wants to support, their hardware inventory, or their software engineering needs.

In the past three decades, our vibrant community has provided many sophisticated and niche systems—using everything from standard software to special-purpose hardware—to fill these needs. For instance, an architect can achieve bandwidth allocation goals with end-to-end congestion control, centralized allocators, traffic engineering, in-network packet scheduling, and various combinations of these options. Architects can deploy firewalls, connection terminating proxies and load balancers at edge sites, near datacenters, or inside servers; these features can be software-based or use hardware accelerators. To process packets, end hosts may use the OS network stack, kernel bypass libraries, or hardware offloads. Each of these choices has different tradeoffs and places different constraints on the rest of the architecture. For instance, deploying a load balancer at an edge site may make it easier to also deploy a firewall there since resources are already provisioned. Likewise, some kernel bypass network stacks require applications to be modified or be compatible.

The result of this new landscape is that network architects must now not only make an increasing number of choices, but also choose between an increasing number of options,

each with its own nuances. The sheer number of choices and the complex interactions between those choices places a large burden on the working memory of human network architects. While hyperscaler organizations may be able to dedicate human resources to the task of designing and maintaining network designs, not all organizations that build complex infrastructure are hyperscalers. Even for hyperscalers, this reliance on fallible humans is likely to cause errors, just as writing complex software in assembly is unsustainable; indeed, hyperscaler organizations have already begun to use computer assistance for keeping track of network hardware inventory and configuration [25, 33]. We propose extending this to consider which systems to deploy as well as what hardware to deploy them on.

While the community invests significant effort into characterizing individual systems by benchmarking their implementations and by simulating and mathematically analyzing their models, there are no principled methods to decide how to put them together into a complete architecture design. Consider an oracle that could model the interactions between an architecture design's component systems and hardware. Architects could ask such an oracle whether a candidate design (i.e, specific choices of hardware and software) would meet their applications' needs, or even ask the oracle to synthesize a good design. Meanwhile, teams of architects could use the oracle to ensure that their design is cross-compatible with other teams' architecture. Further, system and hardware developers can use information the oracle has to configure and develop their systems to better serve the application. Creating such an oracle is difficult, but we take a first step towards it in this paper and call for further research to help navigate the network architecture design tradeoff space.

We posit that, to reason about usefully large architecture design questions, the modelling must be shallower. Since it is quite difficult to entirely understand the behavior of even one individual component, reasoning about components' interactions will be intractable without abstractions. We therefore propose the development of an *automated lightweight reasoning engine* to encode known facts about deployable systems, hardware components, and application workloads, without encoding any information about their implementations.

Such an engine can serve as a machine- and human-readable compendium of knowledge gathered over years of work by the community. Individual facts are often lost in natural-language design documents; encoding them in a reasoning system allows for surfacing them when they become relevant. We envision that, once bootstrapped by a small team, this compendium will receive contributions from the entire community.

In this position paper, we take a first step towards realizing this vision. First, in §2 we argue that network architecture design is an error-prone task that would benefit from computer aids. Second, in §3 we explore design options for realizing a lightweight automated reasoning system, including using large language models to automate knowledge gathering. Third, in §5.1 we provide examples of how it is possible to use lightweight automated reasoning to reason about network architecture design problems.

2 Why lightweight automated reasoning?

We argue that current ad-hoc approaches no longer suffice for modern network design problems, and further that ad-hoc design is to blame for a class of network faults and downtime.

2.1 Growing Search Space

Modern networks require complex functionalities satisfied by multiple systems deployed together performing different tasks (monitoring, serialization, firewalling, congestion control, transport stack). For each task, there are a large number of systems that can perform it. For example, consider the problem of bandwidth allocation. There are many systems that divide available capacity between network participants according to some sharing policy. Even just with datacenter networks, systems' details vary widely, from fully decentralized congestion controllers such as Cubic [16], DCTCP [1] and HPCC [22], to entirely centralized controllers like Fastpass [30] and BwE [20]. Operators specialize for specific parts of the network at varying spatial and temporal scales, and each deployed system plays a part in determining the ultimate bandwidth allocation an individual flow can achieve.

2.2 Subtle Interactions

Not only is the set of systems under consideration quite large, but these systems also interact with each other in numerous ways that are often subtle and hard to remember, frustrating deployment efforts. We categorize these interactions in the following broad ways:

Network requirements. Each deployed system increasingly comes with a set of caveats—i.e., requirements for its deployment environment that must hold for the system to be useful. For example, to be useful, a delay-based congestion control algorithm such as Swift or Vegas [3, 21] cannot compete with a buffer-filling one unless it is deployed as a "scavenger" transport. Even in that case, the architect must ensure that the network queues are sufficiently deep to avoid hurting the non-scavenger flows [36].

Since these requirements are subtle, and don't raise an explicit error when unmet, they often trickle down into production systems. For example, it is well known that PFC

requires an absence of cyclic buffer dependencies, and otherwise can cause deadlock. In a real deployment scenario, Microsoft reasoned that no cyclic buffer dependency should exist in their deployment because of their datacenter's routing configuration [14]. However, they missed that Ethernet packet flooding was already in-place, which broke the routing configuration's invariant, causing deadlocks in their production network. While a PFC expert might have anticipated this problem, such subtle interactions are difficult for others to track. Thus, encoding the nuances and checking them systematically is the best way to eliminate this class of faults.

Cross-team interactions. Different teams work on deploying and testing systems at different layers of the data processing stack, and a given layer often makes assumptions about other layers' behavior. Keeping track of these assumptions is challenging, especially since they cut across multiple layers and teams. For example, a recent VMware incident involved VMs achieving zero throughput, where the root cause was a checksum error due to double encapsulation at different layers in an overlay network [18]. While the information about expected checksum and encapsulation techniques was available in the documentation for each layer's component, to prevent this incident architects would have had to cross-reference documentation between the layers.

Resource contention. For example, one form of interaction is contention for resources (e.g. QoS classes, FPGA gates and memory, CPU cores, etc). Most systems require some resources for their deployment, taking away the coexistence of many systems which contend for the same resource.

2.3 Case Study

We further stress the above points noted (and more) with a case study of the rich trade-off space available for designing a modern cloud network. Afterwards, in §3, we will discuss how to model the architectural design decisions we describe here.

In our example, the architect wants to deploy a machine learning inference application. The architect wants the application to serve requests with low latency, so they want to use load balancing. To ensure network delays do not interfere with the application's low latency, the architect also wants to monitor network queue lengths.

Even this straightforward example contains a nuanced web of interactions between the application's component systems, the hardware resources it uses, and the architect's goals. The architect can consider component systems for one of at least five roles: a network virtualization approach (e.g., OVS [31], Andromeda [5], or a hardware-offloaded approach [8]), a network stack (e.g., Linux, Snap [23], NetChannel [4]), a congestion control algorithm (e.g., Timely [24], Swift [21], Annulus [32], BFC [12]), a load balancing algorithm (e.g., ECMP, VLB, packet spraying), and a network monitoring system with the capability for queue-length monitoring (e.g., Simon [11], Sonata [15], Marple [26]). The architect could additionally consider other component systems (e.g., caches, etc), but even the five we enumerate interact in nuanced ways. The architect must also consider the hardware they will deploy systems and their processing logic on, from the number of CPUs to the types of switches (programmable vs fixed-function) and types of NICs (fixed-function, FPGA SmartNIC, or CPU SmartNIC).

Suppose the architect initially chooses the simplest choices: they use OVS for virtualization, the Linux networking stack (with its default Cubic algorithm for congestion control), ECMP for load balancing, and they forgo network monitoring. They might deploy these systems on fixed-function standard hardware. This solution, while easy to understand, is unlikely to meet the architect's goal of low latency. Thus, the architect must pursue more complicated solutions. The architect might observe that ECMP load balancing can lead to load imbalance, and consider using packet spraying instead. Simultaneously, the architect could deploy one of the queue length monitoring systems mentioned above, or using a kernel-bypass network stack option.

Each of these choices requires the architect to satisfy some dependencies to deploy it. For example, packet spraying requires larger reorder buffers at NICs. Using Annulus for congestion control will improve tail latency, but requires switches to support QCN notifications. Particular switch models may support QCN, but offer lower performance when such features are used in conjunction with virtualization features. Deploying Simon for monitoring latencies requires SmartNICs, but if the architect deploys these SmartNICs, then the marginal cost of deploying other systems using SmartNICs decreases since the systems can share SmartNIC resources. Using NetChannel [4] as the networking stack can support high throughput, but this is only relevant at NIC speeds above 40 Gbit/s; Shenango [28] offers low latencies but less process isolation. None of these facts is particularly complicated to reason about *in isolation*, but they rapidly become overwhelming to remember *all at once* in the context of a particular application and hardware landscape. As described above, changing one aspect of design can have a ripple effect influencing the choices made at every other part of the design. Fortunately, keeping track of a large volume of details is a task computers excel at, as long as we design the right representation. This is the design task we turn to next.

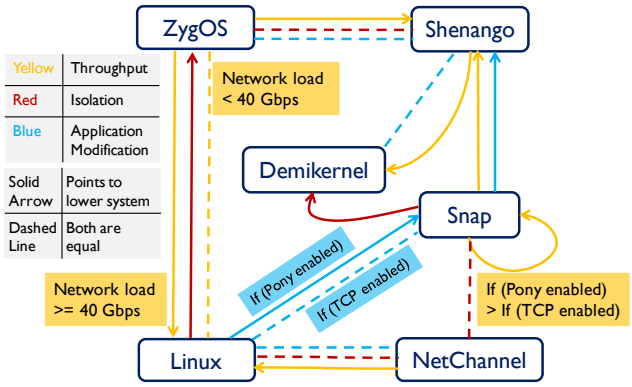

Figure 1: Visualization of a fraction of the partial ordering for network Stacks.

3 Design

3.1 At what level should we encode information?

Our approach hinges on a key observation: architects can check a large number of system properties by focusing on a few crucial details. For instance, the selection between HPCC [22] and Timely or Swift [21, 24] for datacenter congestion control depends, partially, on the fact that HPCC needs INT-enabled switches [22], while Timely and Swift depend on a specific QoS level for acknowledgements and NIC timestamps [21, 24]. This style of reasoning is easy to mimic with a rule-based lightweight reasoning engine. The engine does not need to know what the systems do; it just needs to know how the systems depend on other systems and hardware. Of course, encoding all the hardware components ever designed or all software systems ever developed would be an impossible task. Thus, while any useful reasoning engine should provide *correct* solutions, it must also accept that not all information can be encoded.

Rules-of-thumb. This encoding can thus act as a computer-readable repository of knowledge gained from deployment experience, implementation benchmarks, simulations and mathematical analyses. For instance, consider how one might encode preferences between six different choices for the network stack along three different dimensions: throughput, latency, and application modification. Examining research papers introducing each of the systems, we can easily extract the partial ordering shown in Figure 1. The ordering represents rules of thumb, encoding the relative preference conditional on simple constraints. For instance, Linux is usually sufficiently performant at low link rates (say < 40 Gbps) [23, 28], and while Snap [23] performs better when using Pony, using Pony requires application modification.

Since it is impossible to encode all possible facts, rules-of-thumb will be incomplete. For instance, there is no arrow between Shenango [28] and Demikernel [39] comparing their isolation properties because we couldn't find a comparison in the literature. Indeed, it is impractical to compare every possible pair of systems. Nevertheless, our proposed engine can help architects make more a informed decision regarding whether they should perform an measurement to acquire additional information: it is only needed if the answer changes the final design. For instance, if the architect has a sharp deployment deadline, then using a research system like Shenango is infeasible irrespective of its performance characteristics. This is where it is important to also enable architects to encode their subjective and organization-specific rules. Reasoning about subjective preferences captures the fact that everybody has a different opinion on which is the "best" algorithm or architecture (e.g., the debate on ECN vs delay in datacenter CCAs [40]).

Granularity. Since encoding rules-of-thumb requires the architect's effort, it is not practical to expect them to encode minutiae about each potential system they wish to consider. Rather, we envision that architects will encode rules-of-thumb incrementally, in a breadth-first manner; i.e., architects should start with a minimal set of coarse rules necessary to describe a system before attempting to encode facts at finer levels of granularity (if necessary). While we show in §4 that it may be possible to auto-encode hardware components and software systems using language models, the architect cannot trust the model's output; instead, we envision breadth-first granularity refinement as a strategy to limit the architect's effort in verifying the models' output.

Occasionally, it is possible to make an accurate quantitative statement without having to run a benchmark every time. For example, hardware properties such as the amount of memory, number of ports/queues and various bandwidth measures are easy to accurately characterize. Further, it is common practice to characterize the fraction of CPUs and FPGAs used by individual programs. There are even standardized scaling factors used between different CPU models used in the Linux kernel [38]. On the other hand, because of the highly non-convex optimization space for packet processing pipelines, any description of which subset of P4 programs will fit into a given switch will grow super-linearly (even exponentially!) in the number of programs [9, 10]. One measure of the success of whether this endeavor is whether the length of specification should grow linearly with the number of systems, hardware and workloads included. If not, writing all rules will become very difficult. Therefore, our proposed system will not reason about which P4 programs can fit into switches, except using the type of crude approximations human designers use.

3.2 What should not be encoded?

The example above is one instance of what the reasoning engine cannot reason about. While it is theoretically possible to encode this information, it is not practically scalable to reason about it for the goals that we want to achieve. There are two such categories of facts that we believe are not practical or worthwhile to encode. First, encoding performance numbers is not practical, since they can vary with even small configuration changes, which are beyond the scope of what the engine can capture. Instead, encoding performance as a partial ordering like Figure 1 captures enough information to allow the engine to produce useful outputs. Second, encoding temporal behavior or even the entire algorithm of a software system is again a complex task, and is best left to more dedicated formal reasoning systems. This reasoning engine should only capture the rules-of-thumb information of a system without attempting to encode the entire underlying system itself.

3.3 Who does the encoding?

Who writes these rules-of-thumb? There are too many systems and hardware for any one person to understand all their nuances. However, for any given system there will be one or more experts who do know its implementation details, deployment caveats, and resulting performance impact. A lightweight reasoning framework provides them with an API to document their system's characteristics. To an individual system expert, documenting their system's characteristics is not only straightforward but also independently useful for the same reason documenting code is useful: as a point of future reference.[1] In writing the rules-of-thumb for this paper, we consulted several experts. After bootstrapping with an initial knowledge-base, we hope community efforts will expand both the quality and quantity of the library of knowledge. We additionally explore in §4 whether LLMs can help with automatically encoding knowledge about systems and hardware.

3.4 How do we reason about these rules?

The rules-of-thumb need to be encoded into a logical framework. To this end, we consider leveraging extensive work into rule-based systems [19, 37], SMT solvers [2, 6], and theorem provers [7, 34]. We identify a key tradeoff that will inform this choice of which system of logic we should use. A lot of reasoning can be done with finitely many variables, and the query can be expressed as an existentially quantified formula, for example: "Does there exist a choice of systems such that the following properties and constraints on cost, deployability, etc. are met?". This is a decidable question that a SAT/SMT solver can answer. The power of such solvers to explore combinatorial search spaces will be critical in navigating the intricate design spaces described in section 2.3.

However, there are some properties that cannot easily be expressed in decidable logic. Consider for instance, the rule for checking whether a design using PFC has cyclic buffer dependencies. As noted before, Microsoft unexpectedly encountered this due to ARP flooding in spite of using up-down routing in a Clos topology [14]. If the engine only had a description of the routing behaviors for each protocol, it could in principle, have discovered this bug automatically. However this requires a type of higher order reasoning that is challenging to automate, resulting in undecidable problems if one is not careful in how we encode the problem. On the other hand, an expert might have anticipated this problem with ARP, and could have encoded a requirement that PFC cannot be used with any flooding algorithms. This rule is simple to check with predicate logic. Navigating this tradeoff is space for future work. Our preliminary experience indicates that simple predicate logic can already encode enough rules that it might provide sufficient value without having to venture in undecidable territory.

Finally, we consider whether it is necessary for the reasoning engine to embed "common sense" rules that are already intuitive to human users. For example, it is obvious to any human architect that all servers must use some operating system, but without encoding this fact the reasoning engine could return incoherent results. However, rule-based reasoning typically requires encoding very large libraries of "common-sense" rules, and this could potentially introduce potential encoding and reasoning overhead. While we believe further study is needed to determine the impact of "common-sense" rules, we believe that because (i) our reasoning domain is relatively constrained to just system and hardware deployments, and (ii) the users are experts who possess an intuitive understanding of the "common-sense" rules already, this potential limitation of rule-based reasoning will not have a large impact.

We provide some initial examples of using a SAT solver as the reasoning engine in §5.1.

4 Can We Auto-Generate Encodings?

A primary challenge in applying lightweight automated reasoning is the need to encode various hardware and system instances as deployment candidates. This information is currently distributed amongst natural-language design documents, hardware datasheets, published papers, and the minds of individual architects, and network operators. These need to be encoded in a more formal and structured way. We expect to be able to crowd-source many of these encodings after a small team seeds an initial database of encodings.

[1] Anecdotally, our personal experience indicates that knowing the implementation details of a system does not mean one will *remember* them later.

```
{
    "Model Name": "Cisco Catalyst 9500-40X",
    "Port Bandwidth": "10 Gbps",
    "Max Power Consumption": "950W",
    "Ports": "40x 10 Gigabit Ethernet SFP+",
    "Memory": "16 GB",
    "P4 Supported?": "No",
    "# P4 Stages": "N/A",
    "ECN supported?": "Yes",
    "MAC Address Table Size": "64,000 entries",
    ...
}
```

Listing 1: Excerpted auto-generated encoding for a Cisco Catalyst 9500-40X router.

While humans must lead the charge on this effort, we explore whether it is possible to obtain help from LLMs to generate system and hardware encodings. We note prior work towards a similar goal in the context of hardware components [17]. However, network systems, especially software, are described in a much more heterogeneous document formats (e.g., papers).

In this section, we ask whether (i) LLMs can automatically extract accurate system and hardware encodings from source documents; (ii) LLMs can check candidate encodings against source material for errors; and (iii) finally, whether LLMs can themselves perform lightweight reasoning. In our analysis, we use GPT-4o by OpenAI [27], but we see similar results with other models [35], [29].

4.1 Extracting System Encodings

We find that LLMs perform well when extracting hardware specifications. We described the fields to be filled for switches, servers, and NICs, etc. We provided the spec sheet from the vendor and the LLM extracted the fields with 100% accuracy (unless it was missing in the spec itself). The highly structured and specific nature of the spec sheets was a crucial factor in this. Listing 1 shows part of the output for the Cisco Catalyst 9500-40X switch.

Encoding systems was a trickier affair than this. We started by providing the LLMs some sample encodings describing Systems like Shenango, Sonata, and Simon. We then asked it to create similar encodings capturing all requirements and nuances for Timely, and a few other systems. LLMs were able to identify the hardware requirements of systems, but occasionally missed nuances about how much of a resource is needed, or under what conditions can a system not be deployed. For example, LLMs failed to encode that Annulus [32] is required only when there is competing WAN and DC traffic. In particular, many research papers are written to be largely positive about the systems they propose. Therefore, it was more productive to ask the LLM to find requirements without which the mechanisms paper *cannot* work. Sometimes a sequence of prompts were necessary to answer all questions. Thus, while LLMs can help, for the time being, human supervision is necessary to describe system designs.

4.2 Checking Existing Encodings

Second, and perhaps more realistically, LLMs can *check* rules humans write for 1) completeness and 2) objectivity. While querying LLMs to generate encodings for systems, we found that it does a better job in finding faults in the sample encodings that we wrote by hand. For example, it identified that we missed checking whether the NIC supports interrupt polling, which is a requirement for Shenango. Interestingly, LLMs could not always check for the correctness of a condition (especially if it's loaded with numbers), but they did a better job of checking for the existence of a condition. For example, it does raise an alarm if we encode the wrong number of P4 stages to deploy Sonata. Checking can also improve objectivity. Both humans and the literature are often biased and LLMs can help ensure objectivity. Everybody wants to believe their favorite design is best. In contrast, LLMs can read a broad range of sources (papers, blog posts, bug reports, datasheets etc.) and present any conflicting claim to humans who can thus make a more informed and unbiased decision. LLMs' ability to scan non-primary sources like blog posts and mailing lists will prove to be a valuable asset.

The final design of the reasoning engine must separate the objective properties that are easy for people to agree on (e.g. Shenango dedicates a core for spin polling) and controversial, subjective, questions (e.g. which congestion control algorithm is best in a given setting). Controversial questions can then be annotated by LLMs and humans with links to sources that disagree with what is encoded. In our preliminary investigation, the controversial questions were all about comparisons between systems, while it was easier to objectively encode details about inter-dependencies between systems and hardware.

5 Discussion

5.1 Prototype of the reasoning layer

We attempted to build a shim layer over SAT solvers as an early version of the proposed reasoning layer. We encoded over fifty systems, spread across Network Stacks, Congestion Control, Network Monitoring, Firewalls, Virtual Switches, Load Balancers, and Transport Protocols. In addition, we encode about 200 hardware specs of servers, switches, NICs, etc, from publicly available information. The encodings follow the design decisions discussed in §3.

As an example, Listing 2 shows a simplified version of how we encoded the Simon [11] monitoring system. Line 2 describes the objectives the system can achieve. Lines 3-5 describe the hardware constraints and the resources required

```
SIMON = System(
    solves = [capture_delays, detect_queue_length],
    constraints = And(
        NICs.have("NIC_TIMESTAMPS"),
        computes.cores_needed(CPU_FACTOR*num_flows)))

Ordering(SIMON, monitoring, better_than = PINGMESH)
Ordering(PINGMESH, deployment_ease, better_than = SIMON)
```

Listing 2: System description of SIMON.

```
inference_app = Workload(
  properties = [dc_flows, short_flows, high_priority],
  deployed_at = racks[0:3],
  peak_cores = 2800, peak_bandwidth = 30)

inference_app.set_performance_bound(
  objective = load_balancing,
  better_than = PacketSpray)

Optimize(latency > Hardware cost > monitoring)
```

Listing 3: Description of the ML inference workload in a sample encoding.

to deploy the system. Lines 7-8 compare Simon with other monitoring systems as a partial ordering.

On top of these encodings, we described the case study discussed in §2.3. Listing 3 shows a trimmed version of how we encoded the ML inference application from an architect's POV. Lines 2-4 describe where the application is deployed, what kind of properties does it have, and how many resources does it use. Lines 6 through 10 describe the performance objectives and constraints that the architect attempts to enforce.

We used this encoding to verify how the system deployment changes as we add more workloads with different properties. We formulate the following queries to mimic some realistic situations,

- I want to support more applications, but I can't change my servers since that requires time and human effort.
- I have already deployed Sonata, and I don't want to change it unless there are huge performance benefits or cost savings.
- Given my current workloads, is it worthwhile to deploy CXL memory pooling?

The output of the reasoning layer mimics the outcomes discussed in §2.3.Our ability to encode these questions into simple queries motivates the promise of our reasoning framework as an effective tool for designing architectures.

5.2 LLMs as a reasoning engine

We also explore whether LLMs can also be used here to reason about these rules they helped generate. After describing the case study in §2.3 in natural language, we posed the questions discussed in §5.1 to the LLM. While it accurately determined straightforward requirements such as the minimum number of cores needed to deploy all the workloads and systems, it failed to return correct results when faced with nuances such as comparing the performance of Snap and Demikernel in a given context, or deploying *P4-friendly* systems when forced to use programmable switches. With advancement in LLM intelligence, we presume that they will get better at reasoning about these nuances. But they are currently falling short of giving meaningful results.

6 Future work

Proof modularity. In formal reasoning systems such as proof assistants (e.g., Coq [34], Lean [7]), modifying a system's encoding over time to reflect its evolution can involve a significant amount of work. Instead, we believe that our lightweight reasoning system will support *modular reasoning*. Since we don't assign semantics to any individual property, it is possible for a new system (or a new version of an old system) to update the properties it provides. It is thus useful to study whether systems can modularly encode their properties in this way, and how the reasoning engine should model interactions between various systems as encodings evolve.

Explainability. It is likely that an architect's inputs to a lightweight reasoning system will be under-specified, leaving both the possibility for multiple viable solutions or none at all. If there are no viable solutions, the reasoning framework should tell the architect which of their requirements are in conflict. Further, a future version of the reasoning system should identify a minimal-effort ordering for the architect to provide to make the solution unique. It will also be important for the reasoning system to identify equivalence clas-ses of system deployments, rather than simply returning an arbitrary but compliant solution, as is often acceptable in program synthesis literature [13].

7 Conclusion

As network architectures become more complex, architects will increasingly rely on automated reasoning to check their decisions and raise warnings about their deployments. In this paper, we show the value of automated reasoning and take first step towards this line of research by designing a way of encoding facts about systems at a level that is simple enough to be broad, yet still allow reasoning engines to give useful and surprising outputs.

Acknowledgements

We thank the reviewers for their feedback. We acknowledge support from NSF grants CNS-2403026 and CNS-2212103. Rahul thanks his parents for their sacrifices and support.

References

[1] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM*.

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[3] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*.

[4] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards µs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *SIGCOMM*. https://doi.org/10.1145/3544216.3544230

[5] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Candromeda Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[7] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE*. https://doi.org/10.1007/978-3-319-21401-6_26

[8] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host Sdn in the Public Cloud. In *NSDI*.

[9] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch code generation using program synthesis. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 44–61.

[10] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2023. CaT: A Solver-Aided Compiler for Packet-Processing Pipelines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 72–88.

[11] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2019. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *NSDI*.

[12] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. 2022. Backpressure Flow Control. In *NSDI*.

[13] Sumit Gulwani. 2016. Programming by Examples - and its applications in Data Wrangling. In *Dependable Software Systems Engineering*, Javier Esparza, Orna Grumberg, and Salomon Sickert (Eds.). Vol. 45.

[14] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.

[15] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *SIGCOMM*. https://doi.org/10.1145/3230543.3230555

[16] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (jul 2008), 64–74. https://doi.org/10.1145/1400097.1400105

[17] Luke Hsiao, Sen Wu, Nicholas Chiang, Christopher Ré, and Philip Levis. 2019. Automating the generation of hardware component knowledge bases. In *LCTES (LCTES 2019)*. https://doi.org/10.1145/3316482.3326344

[18] VMWare Inc. 2023. VMware Container Networking with Antrea 1.7.0 Release Notes. https://docs.vmware.com/en/VMware-Container-Networking-with-Antrea/1.7.0/rn/vmware-container-networking-with-antrea-170-release-notes.pdf

[19] Grigoris Karvounarakis. 2009. *Datalog*. Springer US, Boston, MA, 751–754. https://doi.org/10.1007/978-0-387-39940-9_968

[20] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM*. https://doi.org/10.1145/2785956.2787478

[21] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *SIGCOMM*.

[22] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *SIGCOMM*.

[23] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *SOSP*. https://doi.org/10.1145/3341301.3359657

[24] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*.

[25] 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 *NSDI*.

[26] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*. https://doi.org/10.1145/3098822.3098829

[27] OpenAI. 2024. How can I access GPT-4, GPT-4 Turbo and GPT-4o? https://help.openai.com/en/articles/7102672-how-can-i-access-gpt-4-gpt-4-turbo-and-gpt-4o

[28] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *NSDI*.

[29] Perplexity AI. 2024. Perplexity. https://www.perplexity.ai. AI search and chat tool.

[30] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*. https://doi.org/10.1145/2619239.2626309

[31] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open VSwitch. In *NSDI*.

[32] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa H. Ammar, Ellen W. Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. 2020. Annulus: A Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates. In *SIGCOMM*. https://doi.org/10.1145/3387514.3405899

[33] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *SIGCOMM*.

[34] Coq Team. [n. d.]. The Coq Proof Assistant. https://coq.inria.fr/.

[35] Gemini Team. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[36] Michael Welzl and David Ros. 2011. RFC 6297: A Survey of Lower-than-Best-Effort Transport Protocols. (2011). https://www.rfc-editor.org/rfc/rfc6297

[37] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.

[38] Rafael J. Wysocki. 2017. Capacity Aware Scheduling in the Linux Kernel. https://www.kernel.org/doc/html/latest/scheduler/sched-capacity.html

[39] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *SOSP*. https://doi.org/10.1145/3477132.3483569

[40] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 313–327.