KnapsackLB: Enabling Performance-Aware Layer-4 Load Balancing

ROHAN GANDHI, Microsoft Research, India
SRINIVAS NARAYANA, Rutgers University, USA

CCS Concepts: • **Networks** → **Cloud computing**.

Additional Key Words and Phrases: Application performance, load packing, load balancing

ACM Reference Format:
Rohan Gandhi and Srinivas Narayana. 2025. KnapsackLB: Enabling Performance-Aware Layer-4 Load Balancing. *Proc. ACM Netw.* 3, CoNEXT1, Article 3 (March 2025), 20 pages. https://doi.org/10.1145/3709377

Abstract

The layer-4 load balancer (LB) is one of the key building blocks of online services. In this paper, we empower such LBs to adapt to different and dynamic performance of backend instances (DIPs). Our system, KnapsackLB, is generic (can work with variety of LBs), does not require agents on DIPs, LBs or clients, and scales to large numbers of DIPs. KnapsackLB uses judicious active probes to learn a mapping from LB weights to the response latency of each DIP, and then applies Integer Linear Programming (ILP) to calculate LB weights that optimize latency, using an iterative method to scale the computation to large numbers of DIPs. Using testbed experiments and simulations, we show that KnapsackLB load balances traffic according to DIP performance and cuts average latency by up to 45% compared to existing designs.

1 Introduction

A high performance layer-4 load balancer (L4 LB) is one of the key building blocks of online services. Individual services scale by running on multiple backend servers or VMs with unique IP addresses (referred as *DIPs*). The service exposes a small number of virtual IP addresses, termed *VIPs*, to receive traffic from outside the service. The LB receives traffic coming on the VIPs and distributes it across DIPs. Recently, there has been significant interest in improving the scale, availability and cost of the LBs[17, 19, 23, 27, 28, 44, 47, 48, 59, 64].

Ideally, an L4 LB should split the traffic to provide *best performance* (e.g., latency) across the DIPs. To do so, many of the LBs assume that capacity of the DIPs is same or at least known. When the capacity is same, LBs can simply split traffic equally getting uniform performance on all DIPs. Else, it resorts to algorithms such as weighted round-robin or least-connection to load balance the traffic[9, 19, 23, 48] where the weights are set by the operators based on the capacities of DIPs. However, there is a growing trend of DIPs exhibiting capacities that change *dynamically*, especially in virtualized clusters. Prior work has shown that the capacity of VMs with the same type (*e.g.,* D3.medium in Azure) may vary up to 40% due to noisy neighbors: VMs on the same host compete for shared resources such as caches and memory buses, resulting in variation. Recent proposals and new offerings from cloud providers introduce dynamic changes to the number of vCPUs assigned to VMs (detailed in §2). Together, these factors make capacities variable across DIPs and over time.

Authors' Contact Information: Rohan Gandhi, Microsoft Research, India; Srinivas Narayana, Rutgers University, USA.

ACM 2834-5509/2025/3-ART3
https://doi.org/10.1145/3709377

In such a setup where the capacities of the DIPs change dynamically, we find that existing LBs fall short in achieving a goal of providing best performance. We consider well-known LBs and algorithms: (a) HAProxy LB – one of the widely used open source LB with all available algorithms, (b) Microsoft Azure LB. We find that the existing algorithms fall short in splitting traffic conforming to the capacities, resulting in poorer performance on some of the DIPs (§2.1). Interestingly, the *least connections* algorithm, which is specifically built to adapt to variable DIP performance, also fell short. As a result, the requests going to the over-utilized DIPs suffered poor performance (2× higher latency than other DIPs). Lastly, many existing LBs often remove DIPs suffering from poor performance, instead of adapting to their performance.

In this paper, we revisit the first order question on how L4 LBs split traffic. We argue that L4 LB problem should be modeled as a *Knapsack problem*[42] to pack the load to optimize service performance while conforming to static/dynamic capacities of the DIPs. The key challenge is determining how much load is safe to direct to or away from a DIP without adversely impacting service performance.

This paper presents KnapsackLB, a "meta" LB design that enables other L4 LBs to provide best performance. KnapsackLB does not intend to add to the impressive list of existing LB designs. Instead, KnapsackLB enables setting the DIP weights for any LB that can implement weighted load balancing. A significant number of LB systems provide an interface to specify the weights, including AVI[16], Nginx[12], Duet[28], and HAProxy[9]. KnapsackLB frees admins from configuring weights according to DIP performance that is either static or dynamic. In doing so, it leverages the availability and scalability of existing LB designs while endowing them to provide best performance.

At a high level, KnapsackLB decouples weight computation from LBs and does it at a central controller. This enables KnapsackLB to make optimal decisions with a *global view* across DIPs. We use request-response (service) latency as a proxy for the performance[1]. KnapsackLB uses active probing judiciously to learn a *per-DIP weight-latency curve*: a mapping from a weight to the DIP's average response latency, should that weight be applied to that DIP. This mapping is used to drive an Integer Linear Program (ILP) that computes LB weights to pack load into DIPs, while minimizing the average response latency across DIPs. The KnapsackLB controller uses the LB's existing interface to configure DIP weights. In doing so, KnapsackLB does not require any changes to DIPs, clients, or the LBs.

KnapsackLB addresses three key technical challenges to achieve its goals. First, issuing probes naively for the set of all possible weights would require an impractically high number of latency measurements across hundreds of DIPs. KnapsackLB implements just a few measurements (§4.2, §4.3), each adapting from the results of the past measurements, and runs curve-fitting to learn a weight-latency curve that works well across a wide range of weights.

Second, running an ILP that handles hundreds of DIPs with fine-grained DIP weight settings is computationally expensive. KnapsackLB uses a multi-step ILP to substantially speed up the calculation of DIP weights. Specifically, in each step, KnapsackLB runs an ILP that progressively zooms into finer-grained weight settings per DIP, instead of calculating weights in one shot (§4.4).

Third, a weight-latency mapping learned at some fixed load arriving at an LB could become stale. Specifically, the load placed on a DIP, and hence its response latency, depends on both the LB's aggregate incoming load and the DIP's weight. KnapsackLB avoids learning a new weight-latency curve for each aggregate load. Instead, KnapsackLB adapts quickly to variations in traffic rate by rescaling and shifting its learned weight-latency curve (§4.5).

We evaluate a 41-VM prototype implementation of KnapsackLB (§5) on Azure and also at larger scale using simulations. Our results (§6) show: (a) KnapsackLB can adjust the weights as per the

[1] We focus on services where latency matters (e.g., web services).

performance of the DIPs. (b) In doing so, it cuts the latency up to 45%. (c) KnapsackLB adapts to changes in the cluster in terms of capacity, traffic, and failures. (d) KnapsackLB solves ILPs quickly using optimizations. (e) KnapsackLB can work with a variety of existing LBs. (f) KnapsackLB incurs only a small overhead in terms of CPU cores and dollar costs.

This work does not raise any ethical issues.

2 Background

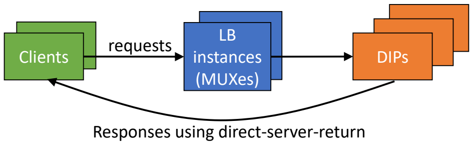


Fig. 1. LBs run on multiple instances called MUXes.

An online service deploys multiple servers (called DIPs) to scale and to provide high availability. The DIPs run behind a load balancer (LB) that exposes one or more virtual IPs (VIPs) to receive traffic from outside the service. In this paper we focus on layer-4 LB that splits the traffic across DIPs using TCP/IP fields. VIPs offer multiple benefits for scalability, security and high availability[48].

As shown in Fig.1, LB designs run on multiple instances (called MUXes). MUXes intercept the VIP traffic and split them across DIPs[23, 48].

DIP selection: MUXes need to process packets with high throughput and low latency. Thus, they pick DIPs for new *connections* using fast algorithms such as (weighted) round robin, hash over TCP/IP fields, power of two or (weighted) least connection[10]. However, a long standing assumption for these algorithms to work well is that the *DIPs have the same or known capacities* (capacity = max. throughput of a DIP) so that MUXes can set weights to get uniform load balancing, not overload DIPs and achieve good performance across DIPs. However, as detailed next, there is a growing trend towards DIPs of the same service to have different/dynamic capacities. LB algorithms fall short in automatically optimizing performance in such cases.

2.1 Limitation-1: LBs Cannot Adapt to Dynamic Capacities

In this section, we describe: (a) why DIPs can have capacities changing dynamically, (b) how existing LBs fall short in providing best performance when capacities of DIPs change dynamically.

2.1.1 DIPs Can Have Capacities Changing Dynamically

Recent works have shown that the VMs in cloud have dramatically different capacities even when they fall in the same VM type (*e.g.,* D3.medium in Azure) (more details in §7). The change in capacities mainly stems from *noisy neighbors, i.e.,* the contention from the VMs on the same host. Even if the vCPUs are isolated, other resources including CPU caches[60], main memory[52, 53], and the memory bus are shared. VMs on the same host contend for such shared resources dynamically, translating to dynamic DIP capacities. Additionally, our private conversation with a major cloud provider also indicated *dynamic over-subscription*, *i.e.,* the number of vCPUs sharing the same physical cores changes dynamically depending on customer demand, causing changes in capacities. Importantly, the *changes in capacity are variable and may occur at any time.* As we show in the next section, such change in capacities impacts the performance, and existing LBs do not react well to the dynamic capacities.

2.1.2 LBs Dont Adapt to Dynamic Changes in DIP Capacities

We consider two L4 LBs: (a) HAProxy[9], a widely used open-sourced LB, and (b) Microsoft Azure.

HAProxy. Fig.2 shows our experimental setup. We have HAProxy running on a 8-core VM. We have three DIPs – DIP-HC (x2) and DIP-LC (HC/LC = High/Low Capacity) on 2-core VMs each. All three DIPs run web servers that compute a cache intensive task for every HTTP request. DIP-HC VMs have the same capacity; we change the capacity of DIP-LC by running varying number of

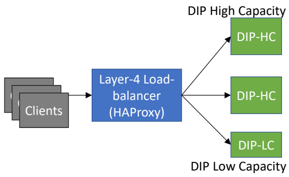
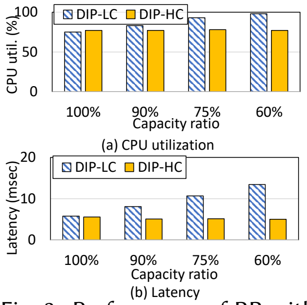


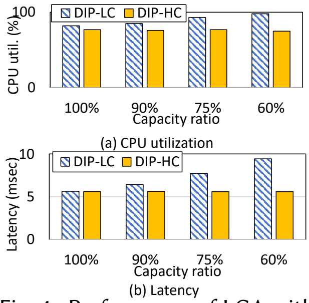
Fig. 2. Setup for evaluating RR and LCA policies in HAProxy LB.



(a) CPU utilization

(b) Latency

Fig. 3. Performance of RR with changes in capacity.



(a) CPU utilization

(b) Latency

Fig. 4. Performance of LCA with changes in capacity.

copies of an antagonist process that thrashes caches (and partially consumes CPU). All VMs run in Azure on DS series CPU running Ubuntu 20.04.

We evaluate all algorithms in HAProxy and present results for 2 popular algorithms – (a) round-robin (RR), (b) least-connection (LCA). RR simply rotates new connections across 3 DIPs. LCA uses the number of active connections as a proxy for load and selects the DIP with the least number of connections when the new connection appears.

Metric of interest: For both policies, the metric of interest is request-response latency (for client requests) for all DIPs. We take the average for 10K requests.

Fig.3 shows the CPU utilization and request-response latency for RR. The X-axis shows the ratio of capacity of DIP-LC to DIP-HC, denoted as "Capacity ratio." When capacity ratio is 100% (all DIPs with the same capacity), the CPU utilization on all DIPs is close to 80%. Next, to emulate the dynamic/different capacity, we deploy the antagonist process. Note that, the impact of noisy neighbors could be variable and unpredictable. We want to evaluate if RR (and LCA later) can adapt to new capacities automatically without manual intervention from the service owners. However, HAProxy continues to split the traffic in the same way as before (equal split). As a result, for lower capacity ratio, as DIP-LC has lower capacity, it hits close to 100% utilization before DIP-HC VMs.

The CPU imbalance also translates in difference in request-response latencies (Fig.3(b)). Because of dynamic reduction in capacities and HAProxy continuing to send traffic at same rate, DIP-LC sees higher CPU utilization and higher latencies for the same incoming traffic rate. The imbalance in latencies across DIPs only gets worse as capacity ratio decreases.

Now we turn to LCA and evaluate whether it can balance load to provide lower latencies. We keep the same setup. Fig.4(a) and Fig.4(b) show the CPU utilization and end-to-end latency as we decrease the capacity ratio. Surprisingly, despite LCA being a policy that considers performance, our observations are similar to RR. As shown in Fig.4(a), as we decrease the capacity ratio, there is imbalance in the CPU utilization on DIP-LC compared to DIP-HC VMs. While the imbalance is slightly smaller than RR, it is still prevalent. LCA also results in CPU fully utilized on DIP-LC whereas DIP-HC VMs are under-utilized resulting in higher latencies on DIP-LC.

At first glance, it may seem that LCA will adapt to different performance by assigning more new connections to the DIP with the fewest connections at that time. A faster DIP will mostly result in fewer connections than the slower counterparts. However, in doing so, it effectively *splits the number of concurrent connections equally* among DIPs. DIPs that finish connections faster will get more connections. However, some DIPs (with lower capacity) can get overwhelmed with many concurrent connections. In such cases, LCA never reduces the concurrent connections on such DIPs. In the previous example with capacity ratio of 60%, DIP-LC got roughly 27% of all connections; all such connections going to DIP-LC overwhelmed DIP-LC and caused higher latency.

The other algorithms also exhibited CPU and latency imbalance (not shown due to limited space).

Limitations using Azure public L4 LB: We repeated the above experiment using Azure L4 LB. We used above 3 DIPs with Azure LB. Azure LB only provides hash onver IP 5-tuple-based load balancing[1]. Table 1 shows the latency and CPU utilization on DIP-HC and DIP-LC. Here DIP-LC has 60% capacity compared to DIP-HC. Unsurprisingly, as IP 5-tuple-based load balancing is not built for best performance, we see imbalance in terms of the CPU utilization and latency – latency of DIP-LC to be 43% higher than DIP-HC.

These results show that HAProxy and Azure LB fail to provide good performance (lower latencies) by automatically adapting traffic based on performance.

Agent-based LB: HAProxy and Azure LB are *agent-less*, i.e., they do not have any agents on the clients and DIPs. There have been *agent-based* works (e.g.,[19]). We contrast against them in §6.5.

Table 1. Load imbalance using Azure L4 LB.

DIPs	CPU utilization	Latency
DIP-LC	84%	7.18 msec
DIP-HC	51%	5.00 msec

2.2 Limitation-2: LBs Cannot Adapt to Differences in Performance

2.2.1 DIPs Can Have Different Performance

Our private conversation with a major online service owner (using more than 100K VMs) indicated that they do not always use pay-as-you-go model where they dynamically scale up/down VMs based on the demand. The key reason is that public cloud providers do not always have capacity needed, especially during peak hours. As a result, the service owners don't release their VMs. Instead, they reassign the VMs to different parts of their service. Such a reassignment results in DIPs of different types (e.g., DS[6] and F[7] type in Azure) for the same VIP, resulting in DIPs with different performance in the same DIP-pool of a service. We observed F-type VMs to provide 15-20% lower latency for simple request-response traffic compared to DS-type VMs in Azure.

2.2.2 LBs Dont Adapt to Differences in Performance

In this experiment, we show that LBs dont react to static differences in performance. We have two DIPs behind LB – one each from DS- and F- series in Azure with same number of cores. Both the DIPs run web server. Again, our metric of interest is request-response latency – whether LBs provide the best overall latencies. However, both HAProxy (using RR) and Azure LB *split the traffic equally among such VMs* that did not yield optimal latency. Ideally, LBs should have sent more traffic to F-type VMs to lower overall latency. Next, HAProxy (using LCA) sent 2% more requests to F-type VMs not leveraging the full potential of F-type VMs. We can further lower the latency by carefully sending more traffic to F-series VM.

2.3 Changing Weights to Adapt to Capacity

HAProxy and Azure LBs did not automatically adjust the traffic split to provide best performance. However, manual intervention by changing the weights to split the traffic is also not trivial. If some DIPs are suffering from poor performance (hotspots), if too little traffic is taken away, that may not clear the hotspots. On the other extreme, if too much traffic is taken away, it risks hotspots moving somewhere else.

3 KnapsackLB

3.1 Goals

We present KnapsackLB, a “meta” LB designed to meet the following goals:

Performance-optimized LB: Our vision is that KnapsackLB should enable other L4 LBs to split traffic to provide the best performance. Admins can plugin any DIP, and leave it to KnapsackLB to provide best performance *without taking any hints about capacities or performance.* KnapsackLB should adjust load even when the performance changes dynamically. In essence, KnapsackLB must free-up admins from the labor of setting up performance-optimized LBs.

Generality: KnapsackLB intends to work with existing LB designs *without any changes to MUXes, DIPs or clients.*

Zero-touch and agent-less design: KnapsackLB should not run any agents on DIPs, clients or MUXes. Consequently, KnapsackLB should work without access to DIP SKU, CPU utilization, or any performance counters on DIPs. This goal helps customers maintain privacy about their resource usage, and helps KnapsackLB reduce deployment overheads.

Everything online: KnapsackLB should not require any offline profiling data. Using just the IP addresses of the DIPs, KnapsackLB should perform all its functions online.

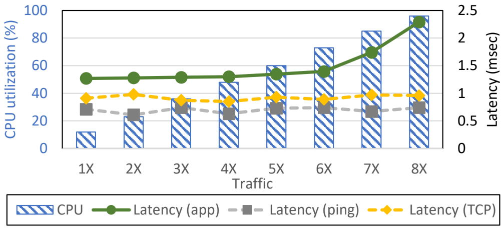


Fig. 5. Impact of increasing weights (traffic) on latency (y2 axis) and CPU utilization (y1 axis). TCP and ICMP pings are unaffected by changing weights (traffic).

3.2 KnapsackLB Overview

KnapsackLB casts the LB problem as a Knapsack problem to pack the load to optimize the service latency (§3.4). We focus on DIPs running services where latency matters (e.g., web services).

KnapsackLB has three key aspects: (a) gauging *DIP performance vs. weights*, through an active probing approach (§4.2, §4.3) that works without DIP, MUX, or client modification or without agents on them, (b) weight computation at a centralized controller using multi-step ILP (§4.4) with the goal of packing load into DIPs to minimize average response latency, and (c) programming the DIP weights at LB controller.

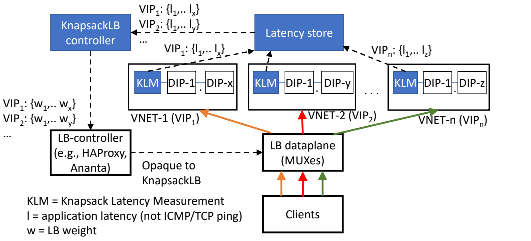

KnapsackLB controller
Latency store
VIP$_1$: {l$_1$,.. l$_x$}
VIP$_2$: {l$_1$,.. l$_y$}
...
VIP$_1$: {l$_1$,.. l$_x$}
VIP$_n$: {l$_1$,.. l$_z$}
KLM
DIP-1
DIP-x
KLM
DIP-1
DIP-y
KLM
DIP-1
DIP-z
. . .
VIP$_1$: {w$_1$,.. w$_x$}
VIP$_2$: {w$_1$,.. w$_y$}
...
VNET-1 (VIP$_1$)
VNET-2 (VIP$_2$)
VNET-n (VIP$_n$)
LB-controller (e.g., HAProxy, Ananta)
Opaque to KnapsackLB
LB dataplane (MUXes)
Clients
KLM = Knapsack Latency Measurement
l = application latency (not ICMP/TCP ping)
w = LB weight

Fig. 6. KnapsackLB architecture. Blue boxes denote KnapsackLB components. Solid and dotted lines denote data traffic and control signals respectively.

Using weights to control traffic: We observe that many existing LB designs provide an interface to specify the weights for splitting the traffic [9, 12, 16, 28] (note that these LBs do not automatically calculate weights as shown in §2). KnapsackLB uses this interface to program the weights. This way, KnapsackLB can support a variety of other LBs. Wherever such an interface is not provided, we use DNS based LB (§6.6). Importantly, the MUXes themselves remain unmodified and their throughput and latency are unaffected by KnapsackLB.

Application latency as a proxy for performance: KnapsackLB optimizes for the *application-level latency* using requests that are served by the service running on the DIPs. We use such a latency as a proxy for service performance. As shown in Fig.5, ICMP and TCP pings (using SYN/SYN-ACK) do not reflect the load on the DIP as they are handled by the OS. In contrast, application (app) requests are handled by the service on the DIPs and reflect the performance of the service including queues formed at applications.

An alternative approach is to monitor CPU usage on DIPs and steer traffic away from hot DIPs. However, it will fall short as: (a) it either requires agents on the DIPs or access to client subscription – both raising privacy concerns and are non-goals for KnapsackLB. (b) application latency is not a function of just CPU utilization. Application latency also depends on many factors across the stack such as CPU cache at hardware[39], throttling at hypervisor and application factors (such as queues). It is not trivial to capture multitude resource counters and translate to application performance[39]. Instead, measuring application latency directly captures the service performance, and eliminates the need to capture and translate multitude signals from hardware to software.

3.3 Architecture

As shown in Fig.6, KnapsackLB has three loosely coupled components: (a) KLM (KnapsackLB Latency Measurement), (b) latency store, and (c) KnapsackLB controller.

KLM resides in each customer-level virtual network (VNET). For simplicity, we assume there is one externally-visible VIP per VNET. KLM periodically measures the latency for requests from each DIP. KLM directly measures the latency at the service-level (*e.g.,* HTTP requests) using service URLs provided by the administrators. KLM also sends requests directly to the DIPs (using IPs of DIPs; bypassing MUXes) to eliminate the interference of the MUXes on DIP latency. The latency store persists the measurements from the KLM. Implementing measurements that directly measure application-layer latencies and storing them separately obviates the need for any instrumentation on DIPs, MUXes or clients, and enables KnapsackLB to avoid changes to the MUXes, DIPs and clients, broadening KnapsackLB's utility to a variety of LBs and settings.

The KnapsackLB controller consumes DIP latencies from the latency store. It computes the weights using Integer Linear Program (§3.4) and sends them to the LB controller (e.g., Ananta, HAProxy), which in turn programs the MUXes with the new weights. The MUXes only need to store the weights and may implement the weights using WRR (weighted round robin). Many LBs including Ananta, Duet, HAProxy, Nginx, Maglev and Beamer support this, although the latency to change the weights at MUXes does differ across LBs. Existing LB controllers (Fig.6) are *not* on the critical path of the traffic[23, 48]. Likewise, *none of the* KnapsackLB *components are on the critical path.* KnapsackLB does not cause any performance degradation when MUXes handle packets.

Discussion on heterogeneous requests and URL at KLM: DIPs can handle different types (heterogeneous) of requests differently (e.g., through cache or disk or other microservices). Similarly, a DIP could run different operations. E.g., running long "SCAN" or short "GET" requests in the context of key-value stores. Irrespective of the operation, if a DIP is overloaded, it will build the queues for *all* requests and it will be reflected on the requests sent by the KLMs using the URLs set by admins. It may happen components *other than DIPs* (e.g., backend database) become bottleneck that inflates latency. Even then, KnapsackLB will set the weights to optimize for latency.

3.4 Framing Load Balancing as a Knapsack Problem to Optimize Service Latency

ILP Variable: $X_{d,w}$

Objective: Minimize $\sum_{d \in D} \sum_{w \in W_d} X_{d,w} \cdot l_{d,w}$

Constraints:

Only one weight for each DIP: $\forall d \in D, \sum_{w \in W_d} X_{d,w} = 1$ (a)

Total weight is 1: $\sum_{d \in D} \sum_{w \in W_d} X_{d,w} \cdot w = 1$ (b)

Allowed imbalance: $y_{max} - y_{min} \leq \theta$ (c)

$\forall d \in D, y_{max} \geq \sum_{w \in W_d} X_{d,w} \cdot w, y_{min} \leq \sum_{w \in W_d} X_{d,w} \cdot w$ (d)

Fig. 7. ILP formulation.

Notation	Explanation
Input	
D	Set of DIPs
w	Weight between [0,1]
W_d	Set of weights for d-th DIP
$l_{d,w}$	Latency on d-th DIP for w-th weight
y_{max}, y_{min}	Max. and min. weights across DIPs
(output) $X_{d,w}$	(binary) Set if w-th weight is assigned to d-th DIP

Table 2. Notations used in the ILP.

Rather than a "typical" LB approach of spreading load evenly, we frame LB as a Knapsack problem[42] to pack the load as per DIP capacities to *minimize overall service latency* (measured by KLMs above). To do so, we present an Integer Linear Program (ILP) that calculates weights to assign to DIPs to optimize for service response latency. The ILP is shown in Fig.7. The notations are listed in Table 2. There is a distinct ILP that assigns weights for each VIP.

Rather than letting DIP weights vary arbitrarily in the interval [0, 1], KnapsackLB uses a fixed set of possible weights that may be assigned to any DIP. The ILP variable $X_{d,w}$ (boolean) indicates if the w-th weight is assigned to the d-th DIP. The objective is to minimize the total mean latency

(Fig.7)[2]. Using a discrete set of weights (*e.g.,* $W_d = \{0.1, 0.2, ...0.9, 1.0\}$) allows us to provide $l_{d,w}$, the response latency at the d-th DIP if it is assigned the w-th weight, as a numerical parameter to the ILP. The values for $l_{d,w}$ are computed before running the ILP (§4.2).

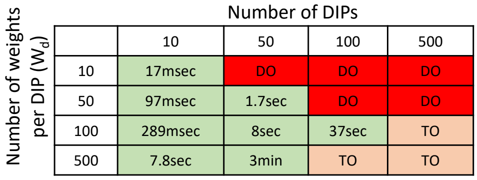
There are four constraints: (a) We assign only one weight to each DIP, (b) the sum of total weight assigned across all DIPs is 1. (c) the imbalance is restricted to θ (we do not aim to load balance equally), (d) we specify y_{max} and y_{min} as the maximum and minimum weights across all DIPs.

Knapsack problems in general are NP-complete[32]. We choose ILP approach as off-the-shelf ILP solvers optimize for a given objective, and have long been used for resource allocation problems[29, 38]. However, in this paper, we show the challenges in using such ILPs for LB (§3.5) and our design choices to overcome those challenges (§4.1).

3.5 Technical Challenges

There are two key challenges in implementing load balancing using the above ILP formulation.

The first challenge is collecting the $l_{d,w}$, *i.e.,* the latency for d-th DIP for w-th weight. As we do not know the latencies (performance) beforehand, a strawman approach could be to measure latency uniformly for w in [0,1]. However, such an approach would require latency measurements at large numbers of weights. Imagine two DIPs with capacity 1× and 19× with roughly equal latency when not loaded. The optimal weights for such a case are 0.05 and 0.95. Simply measuring latencies at 11 points in range [0,1] (0,0.1,0.2, ..., 1) will not yield an optimal split. Worse, with these W_d, the ILP can only calculate weights where at least one DIP is overloaded. Thus, we need latency measured at weights with finer resolution. However, increasing the resolution also increases the number of latency measurements required per DIP. Further, since DIPs can have different performance, we might not be able to reuse the measurements from one DIP to another. For 100s of DIPs and measurements at 100s of weights/DIP can make this approach prohibitively expensive.



Number of DIPs

Number of weights per DIP (W_d)	10	50	100	500
10	17msec	DO	DO	DO
50	97msec	1.7sec	DO	DO
100	289msec	8sec	37sec	TO
500	7.8sec	3min	TO	TO

Fig. 8. ILP performance for varying #DIPs and #weights per DIP. DO and TO indicate DIP Overload and Timeout (ILP could not finish in 20 mins).

The second challenge is the computational feasibility of the ILP even when all the measurements $l_{d,w}$ are available. We run an experiment when all DIPs have the same performance. Fig.8 shows the time to compute the weight assignment for varying number of DIPs and weights per DIP (W_d). The weights are chosen uniformly between [0,1]. Even for a single VIP of 500 DIPs, the ILP results in DIP overload (DO) (load for at least one DIP is above its capacity) or timeout (TO) (ILP takes 20+ mins). Timeouts are especially worrisome as they affect the system's responsiveness to failures and traffic dynamics (§4.5).

4 Detailed Design of KnapsackLB

4.1 Key Algorithms

KnapsackLB includes five algorithmic components to address the challenges in §3.5.

C1. Curve fitting to reduce the number of latency measurements: We observed that we can make-do with a small number of latency measurements and use polynomial regression to build the curve and estimate latency at other weights (§4.2).

C2. Adaptive weight setting based on prior latency measurements: Complimentary to C1, our second algorithm adaptively determines the next weight to use to conduct a latency measurement, given existing weight-to-latency measurements. The algorithm is loosely inspired by TCP congestion control. Together, C1 and C2 obviate the need for hundreds of measurements per DIP (§4.3). As shown in §6, KnapsackLB works with fewer than 10 measurements per DIP.

[2] The objective can be easily changed to other objectives such as minimize max. latency or minimize sum of max. latency.

C3. Multi-step ILP computation: We need to decide on the set of weights to feed to the ILP (W_d). Rather than feeding a large set of possible weights in one shot to the ILP, we feed the weights in multiple steps, with increasing resolution on the weights, while holding the size of W_d constant (§4.4), which substantially reduces running time without affecting accuracy.

C4. Quick reaction to dynamics: Online services exhibit many dynamics including traffic changes, failures, and capacity changes. We design mechanisms to quickly react to dynamics (§4.5).

C5. Scheduling measurements: Even when the weights desired for latency measurements are available (C2), we cannot assign those weights to DIPs in a single shot. We show how to *schedule* latency measurements (§4.6).

4.2 Curve Fitting

We get the values of $l_{d,w}$ with latency measurements at a small number of weights, and implement *curve fitting* (x-axis = weight, y-axis = latency) using polynomial regression. This way we can estimate the latency for other values of weights where we did not make latency measurements directly. Fig.5 earlier shows the increase in CPU utilization and latency as we vary the weights (traffic). It can be seen that the latency increase is minimal at low weights as there is available capacity to handle the requests. We see higher increase in latency as weights increase. For extreme weights, we also observe packet drops (not shown) as there is no capacity left. This latency-weight relationship helps us do curve fitting using polynomial regression of degree two (§6).

4.3 Adaptive Weight Setting for Latency Measurement

Algorithm 1 Algorithm to calculate weights for latency measurement

```
INPUT: $l_0, l_w, w_{now}, w_{prev}, w_{max}^{prev}$
OUTPUT: $w_{next}, w_{max}, isExplorationDone$
1: if $w_{now} - w_{prev} \leq D$ then
2:     $isExplorationDone \longleftarrow 1$; return
3: end if
4: if !(packet drop) then
5:     $w_{max} = max(w_{max}^{prev}, w_{now})$
6:     $w_{next} \longleftarrow w_{now} + w_{now} \cdot \alpha \cdot \frac{l_0}{l_w}$        ▷ Run
7: else
8:     $w_{next} \longleftarrow \frac{w_{now}+w_{prev}}{2}$        ▷ Backtrack
9: end if
```

The goals of the measurement phase are twofold: (a) identify a small number of weights to perform latency measurements, so that the measurements finish quickly and yet provide good curve-fitting, (b) get a rough estimate for the capacity of the DIP in terms of the max. weight (to calculate W_d). Recall that we do not know the capacities of the DIPs beforehand.

Our algorithm for weight selection is inspired by TCP congestion control and has two phases: (a) run, or (b) backtrack, depending on the measured latency increase and packet drop. Algorithm 1 shows the algorithm to calculate weight for each DIP in each iteration in the measurement phase.

The input to the algorithm includes w_{now} and w_{prev}, the weights for current and previous iterations. w_{max} indicates the maximum weight observed so far without packet drop. w_{max}^{prev} is max. weight till last iteration. The input includes l_0 and l_w, the latencies when the weights are 0 and current weight respectively. We measure l_0 when the DIP is newly added by setting its weight to 0. The output includes: (a) w_{next} the weight whose latency should be measured in the next measurement, (b) *isExplorationDone*, a boolean indicating if the computation to get the weight-latency curve for this DIP is over, and it is ready for the ILP to assign the weights for this DIP, (c) w_{max}.

As noted on line 1-2, if the difference between w_{now} and w_{prev} is small (D = 5% of w_{now}), we set the *isExplorationDone* flag. Next, when there is no packet drop, it indicates that there is still some capacity remaining and we can increase the weight. We update the w_{max} (line 5) and increase the weight *proportional to latency* (line 6). When l_w is comparable to l_0, it indicates there is more capacity left, and we can have bigger increase in w_{next}. When l_w is considerably higher than l_0, it indicates we are reaching capacity, and we slow down the increase in w_{next}. α indicates the pace of increase (set to 1 in KnapsackLB). When there is a packet drop (we have reached capacity), we reduce the w_{next} to the average of w_{now} and w_{prev} (line 8) and continue the search. To improve exploration time and reduce packet loss, we assume "packet drop" is *true* on lines 4 and 7 when l_w is 5× l_0 based on the observation that latencies are 5× l_0 or higher when loaded. Lastly, we build the curve quickly so that the traffic change during curve building (few minutes) is usually small[50].

4.4 Multi-Step ILP Computation

In steady state, once the weight-latency curve is available, we need to decide weights in W_d. Once we decide weights, polynomial regression quickly returns corresponding $l_{d,w}$ for the ILP. As shown in Fig.8, the ILP running time increases rapidly with the number of weights. Instead of running the ILP in one step, we run it in two steps, while providing a small number of weights in each step. E.g., instead of running ILP with 100 weights for every DIP, in the first step, we only provide 10 values uniformly in $[0, w_{max}]$ (note, not [0, 1]). This provides a coarse estimate for the weights without packet drop. In the second step, we calculate the weights more precisely. If w_d is the weight chosen by ILP for the d-th DIP in first step, we provide 10 values uniformly between $w_d - \delta$ to $w_d + \delta$ (δ = 10% of w_{max}). We do multi-step iteration only when the #DIPs ≥100. Otherwise, we do the first step only. We program the LB dataplane only after the completion of both steps.

4.5 Handling Service Dynamics

We present mechanisms to address the drift in weight-latency curves over time.

Addressing change in traffic: When total traffic increases, the traffic volume going to the DIPs will increase for the same weights resulting in higher latency for the same weights. We detect the traffic change when we see latency increase for most/all DIPs even when the weights are unchanged. We then "shift" the weight-latency curve "to the left" as follows: let's say the latency was 5 msec at weight 0.5 (w_1). With increased traffic, latency has increased to 7 msec for the same weight. To calculate the new weight, we multiply the existing weights by δ. Let's say the weight (w_2) for latency of 7 msec was 0.625. We calculate $\delta = \frac{w_1}{w_2}$. We multiply all the weights with δ. Similarly, when the traffic has reduced, we increase the weights using above mechanism. We then run ILP.

Addressing changes in capacity: The capacity of the DIPs can change dynamically (*e.g.*, due to a change in co-located VMs). We detect capacity change for a DIP if observed latency differs from the estimated latency by more than a threshold (set to +/- 20% of $l_{d,w}$) – e.g., if the latency has changed from 5 msec to 7 msec. We use above design to calculate new weights and run ILP.

Addressing DIP failures: KnapsackLB detects *application* failures on DIPs when we fail to get successful responses for KLM probes. However, we will continue to send user traffic to failed DIP till the DIP is taken offline. For fast response to failures, we send 3 lightweight HTTP requests every 100 msec. Once the failed DIP is detected, we simply rerun ILP without that DIP and program new weights. In contrast, Azure takes around 10 seconds to take the failed DIP out of the DIP-pool.

Refresh map: We periodically refresh the weight-latency curve. At any point, we limit the fraction of DIPs under refresh to 5% of the total capacity. Note that DIPs may run at lower utilization during refresh as we may try smaller weights than what the DIPs can handle. To ensure enough capacity at all times, we choose max. 5% capacity for refresh at a time. To refresh, we simply measure the latency for the weights as calculated in §4.3. We recalculate the weights for a VIP using the new weight-latency curve as the curve is updated for any of its DIPs.

4.6 Scheduling Measurements

In KnapsackLB, the weights for the DIPs with *isExplorationDone* unset are calculated using algorithm from §4.3. However, we may not be able to measure latency for the calculated weights right away, as the sum of all DIP weights for a VIP needs to be 1. E.g., w_{next} for 2 DIPs can be 0.7 each in one iteration. In such scenarios, we need to *schedule* the DIP weight in multiple rounds.

We classify DIPs with new weights to be scheduled into 3 priority classes: (a) weights for over-utilized DIPs (DIPs with high latency), (b) weights for remaining DIPs, (c) weights during refresh. Within a class, we use FIFO. To schedule the DIPs that have new weights assigned, we use a simple greedy algorithm where we arrange all DIPs according to their priority. We hop over the list of DIPs until either: (1) the weight of DIPs scheduled is 1, or (2) we exhaust all DIPs.

Lastly, we need to wait until the new weights take effect in the dataplane as detailed in §A.

5 Implementation

KLM: We have a VM image running for KLM, which can be deployed in each VNET. The VNET admin sets the list of DIPs and application URL in a config file on KLM. KLM measures latency for every DIP in that VNET every 1 second using the application URL (from admins), and reports average latency over 20 requests (independent of DIP size). We do not consider higher percentiles (such as P90 or P95) because we observed high and variable latency at high percentiles even when the load is not high. Also, we found that average latency correlates better with load (Fig.5).

Latency store: KLMs write the latency to latency store. We use Redis[14] for latency store for its in-memory caching and fault tolerance. The key is VIP and value is list of `<DIP,latency,time>` tuples. The latency store runs in the same datacenter as KnapsackLB controller.

KnapsackLB **controller:** KnapsackLB controller is the heart of KnapsackLB. It consists of modules to (a) calculate weights for latency measurements, (b) scheduling the latency measurements, and (c) running ILP to calculate optimal weights. All the modules (for a VIP) run on the same VM. §6.8 provides the overheads for running the controller for multiple VIPs.

The ILP uses the open source ILP solver COIN-OR[5], written in C++ with PuLP bindings[13]. Together, KnapsackLB uses 4K+ LOC using high-level languages.

6 Evaluation

Table 3. VM details used in evaluation.

DIPs	DIP-1 to DIP-16	DIP-17 to DIP-24	DIP-25 to DIP-28	DIP-29, 30
VM type	DS1v2	DS2v2	DS3v2	F8sv2
#vCPUs	1	2	4	8
#VMs	16	8	4	2

We evaluate KnapsackLB using testbed experiments and simulations for large number of DIPs. Our experiments show that (a) KnapsackLB builds the weight-latency curve fast and requires few points to build the curve; (b) KnapsackLB is able to compute weights optimally using the ILP to minimize the overall latency *without any hints* about the capacity or performance. This is particularly useful as it allows users to add DIPs of any capacity; (c) KnapsackLB substantially improves the latency compared to (weighted) LB policies – used in production systems; (d) KnapsackLB handles the dynamics such as failures, traffic and capacity changes well; (e) KnapsackLB can work with other LBs with and without interface to program weights; (f) KnapsackLB incurs very small overhead in terms of extra resources and costs.

Setup: Our setup consists of 41 VMs running in Azure datacenter. There are 30 DIPs with different capacities (Table 3), one 8-core VM running HAProxy and 8 VMs as clients. Lastly, we have one VM each for KnapsackLB controller and KLM. We use Redis cloud offering[2]. The DIPs run web server doing cache intensive calculation for client requests. The clients: (a) send the requests to DIPs through LB (Haproxy or Azure), (b) measure the end-to-end latency. All VMs run Ubuntu 20.04. As shown in Table 3, we use DIPs of 4 different types. We specifically use VMs across different series (DS and F). The F-series VMs[7] are supposed to be faster than DS-series VMs[6] by up to 2×. However, our measurements found that F-series VM is 15-20% faster than corresponding DS-series VM for our workload. This also highlights that it is not trivial to calculate the weights just by considering number of cores and clock speed. A system like KnapsackLB can help automatically calculate and adjust weights to optimize performance. We set the traffic to 70% of total capacity. We set $\theta = \infty$ in Fig.7 to not put any restriction and optimize for latency.

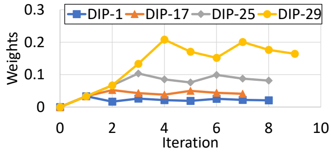


Fig. 9. Weights used for latency measurements.

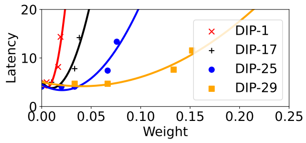


Fig. 10. Curve fitting using polynomial regression.

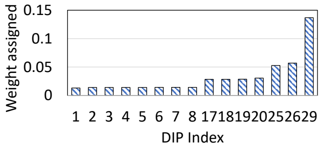


Fig. 11. Weights calculated by ILP.

Baselines: We compare KnapsackLB against HAProxy using Round-robin (RR), Least-Connection (LC) explained in §2.1.2. We also consider Power-of-2 (P2)[46] that randomly picks two DIPs and then selects one DIP with the smaller number of connections among the two DIPs. P2 is also used by Microsoft's YARP[11]. We consider both weighted and unweighted versions. We also consider Azure LB as a baseline.

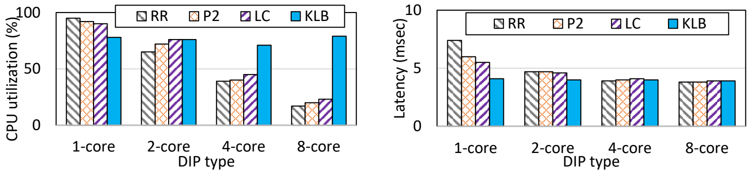
6.1 Weight Assignment in KnapsackLB

Calculating weights for latency measurements: We start our evaluation by calculating the weight-latency curve for all the 30 DIPs. We first start by measuring the latency when weight is 0 and equal (0.033 in this experiment). We then adjust the weights using algorithm 1.

Fig.9 shows the different weights calculated for 4 different types of DIPs. We randomly chose one DIP from each type (Table 3). It took 8-10 iterations to build weight-latency curve for all DIPs. Here, we only show the weight calculated by the Algorithm 1, and not by the scheduler (§4.6). Thus, the total weight per iteration is not 1. Next, for each iteration, the weights are scheduled in multiple rounds. The scheduler took 1.7 rounds for each iteration (on average). Each round was 10 seconds. Therefore, the adaptive weight setting finished in less than 3 mins. The weights calculated for measurements vary across DIP types. Also, the w_{max} for the 4 DIPs calculated are 0.02, 0.04, 0.085, 0.165. Note, w_{max} corresponds to max. weight *without* packet drop. Thus, the values are lower than the peak weight calculated.

Weight-latency curve using polynomial regression: After we measure the latencies for different weights, we use polynomial regression to fit the curve so that we can estimate the latencies for weights not used in measurements. For polynomial regression, we only use the data for which there was no packet drop. As a result, there are only 4 points for DIP-1 to DIP-28 and 5 points for DIP-29,30. Fig.10 shows the weight-latency curve for the 4 DIPs of 4 different types. The points show the actual measurements, while the lines show the curve calculated by polynomial regression. It can be seen that the regression fits the curve well even when it has only a small number of points. As we increase the weights, we expect the latency to go up. However, as can be seen in the Fig.10, the latency may not increase monotonically using regression. To address this limitation, we change the regression output to increase monotonically by setting the latency for a given weight as max. of its latency (by regression) and latency at the previous weight (not shown in figure).

ILP calculation: Next, we calculate the weights using ILP described in §3.4. Fig.11 shows the weights calculated for the 15 DIPs (we select 50% DIPs from each type). The weights are in ratio 1:2:3.9:9.7. ILP assigned more weight to the VM with more capacity (8 core). KnapsackLB optimizes for *total latency* and it got better performance on such 8-core VMs.



(a) CPU utilization (%)

(b) Latency

Fig. 12. Average CPU and latency using testbed with 30 DIPs.

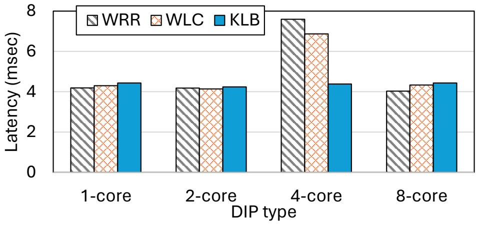


Fig. 13. Average latency using weights for 30-DIP cluster. 8-core VM is F-type. KLB = KnapsackLB.

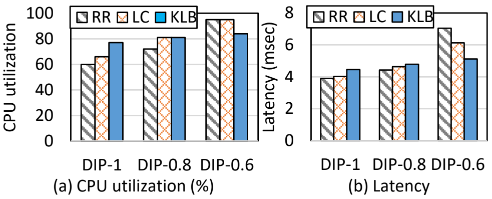


Fig. 14. Average CPU and latency using 3 DIPs of capacity 1×, 0.8× and 0.6×. KLB = KnapsackLB.

6.2 Comparing with Other LB Policies

We now show the improvements in KnapsackLB compared to baseline policies including LC, RR and P2 and Azure (explained in baselines in §6) – both unweighted and weighted versions. The metrics of interest are end-to-end latency observed by clients and CPU utilization on the DIPs. We show the average for 100K requests. We use two DIP-pools: (a) same DIP-pool of 30 DIPs as before. (b) smaller DIP-pool of 3 nodes from §2.1.

30 node DIP-pool with no weights: Fig.12(a) shows the CPU utilization across all DIP types for RR, P2, LC and KnapsackLB (KLB). Unsurprisingly, as RR is not optimized for performance, it results in high CPU utilization for DIP-1 to DIP-24. Conversely, DIP-25 to DIP-30 show lower CPU utilization as they receive less traffic relative to their capacity. P2 and LC improve CPU utilization compared to RR by sending fewer connections to overloaded DIPs but still result in higher CPU utilization for DIP-1 to DIP-16.

Fig.12(b) shows the latency across the 4 types of DIPs. In RR, we observe higher latency for DIP-1 to DIP-24 as such DIPs are overloaded. In contrast, latency from DIP-25 to DIP-30 is low as such DIPs are underloaded. P2 and LC improve the latency but still result in high latency due to overloaded DIP-1 to DIP-16. In KnapsackLB, as the DIPs are not overloaded, we observe low (and uniform) latency across all DIP types. This experiment shows that KnapsackLB cuts the latency by up to 45% compared to RR, and up to 23% compared to LC.

Improvement over Azure LB: As mentioned previously, Azure L4LB only supports equal LB through hash over TCP/IP fields. It assumes equal weight for all DIPs, which is problematic as we have DIPs with different capacities (number of cores). As a result, due to equal splitting, requests going to DIPs with smaller number of cores observe higher latencies. We found that KnapsackLB improves the latency by up to 41% compared to Azure LB.

30 node DIP-pool with weights: In the previous experiment, we did not use the weights. However, service operators can set the weights using hardware properties. In this experiment, we set the weights for RR and LC policies (denoted as WRR and WLC) in proportion to the number of cores. Recall that KnapsackLB does not require such information apriori. Fig.13 shows the average latency across all DIP types for WRR, WLC and KnapsackLB (KLB). We found that the throughput of 4-core DS-type VM did not scale linearly with number of cores (8-core F-type VM also did not scale linearly but in general had more capacity). As WRR is not performance aware, its traffic to 4-core DIPs was unabated, causing the DIPs to be overloaded and caused high latency. WLC reduced the traffic to such DIPs to a small extent. In contrast, KnapsackLB reduced the traffic to such DIPs lowering latency. Compared to WRR and WLC, KnapsackLB reduced the latency by 42% and 36.2% on such DIPs.

3 node DIP-pool with weights: Next, we measure the latency and CPU utilization based on the DIP-pool from §2.1 running on 1-core VMs. However, we change the capacity to emulate noisy neighbors – we use capacities of 1×, 0.8× and 0.6×. In this experiment, we use weighted RR and LC with weights set to number of cores (1:1:1). Note that, the impact of noisy neighbors could be dynamic, variable and unpredictable. As shown in Fig.14, both (weighted) RR and LC fall short in load balancing the traffic as per capacities, and over-utilize DIP-0.6 (DIP with capacity 0.6×) while there is capacity available on other DIPs. As a result, we also observed higher latency on DIP-0.6.

In contrast, KnapsackLB (KLB) substantially improves the load balancing. It reduces the CPU utilization on the DIP-0.6 while making use of the available CPU on DIP-1. As a result, we observed uniform CPU on all the three nodes. In doing so, KLB improved the latency for all the connections that RR and LC sent to DIP-0.6. Compared to RR and LC, KLB cut latency by up to 37% and 29%.

6.3 Dissecting the Gains through Latency and ILP

KnapsackLB is based on two key ideas: (a) using application latency for load balancing, (b) packing the load through ILP. In this section, we show the gains through individual ideas.

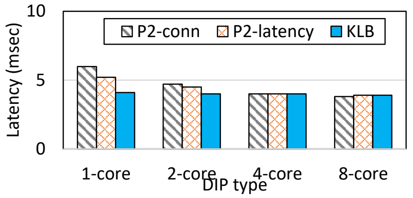


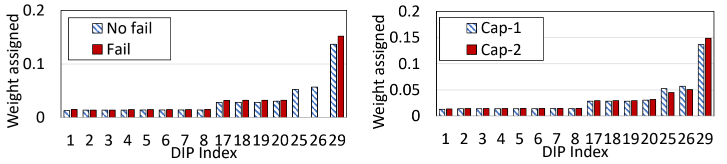
Fig. 15. Dissecting the gains in KnapsackLB across application latency and ILP.

To do so, we build a baseline (P2-latency) where we assign the DIP using power-of-2 combined with application latency (unlike power-of-2 in the previous section that used number of connections). It randomly picks two DIPs and chooses a DIP with smaller application latency. Recall that, by default, KnapsackLB probes the latency every 1 second which is too late for P2-latency baseline. For this experiment, we reduce the latency probe to 100 msec and send the requests from clients when latency probes finish. Note that, we only do this to measure the gains from just using application latency.

We use the 30-node DIP pool from previous section, and also run P2 from previous section that uses number of connections (P2-conn). Fig.15 shows the latency for using P2 with latency (P2-latency) as well as P2-conn and KnapsackLB. It can be seen that P2-latency performs better than P2-conn but is considerably worse than KnapsackLB. This experiment shows that: while using application latency helps, it helps more when combined with ILP based packing in KnapsackLB.

ILP with number of connections and CPU load: we also evaluate an ILP with different objectives: (a) minimize max. number of connections at DIPs. Such a method is lucrative as it is agent-less (no agents on DIPs). However, it converged to splitting number of connections equally among DIPs (similar to LCA) which suffered the same limitations mentioned in §2.1.2, (b) minimize max. CPU load. This method requires agents on the DIPs to gather CPU utilization. However, as it does not consider latency, it suffers the same limitations as §2.2. Being agnostic to performance, it sent fewer connections to 8-core (F-series; faster) VMs. Compared to KnapsackLB, such a design results in higher latency up to 16%.

6.4 Handling Dynamics



(a) Weight change due to failure of DIP-25,26

(b) Weights change as capacity changes for DIP-25 to -28

Fig. 16. Handling dynamics in KnapsackLB.

We now show the change in weights as we address dynamics due to: (a) failures, (b) change in capacity and (c) change in traffic. We use the DIP-pool with 30 DIPs from Table 3.

Addressing failures: In this experiment, we fail DIP-25,26 while keeping the traffic unchanged. Fig.16(a) shows the weights before and after the failure. We observed that the weight of the failed DIP *was not* equally split among other DIPs. DIP-1 to DIP-16, and DIP-17 to DIP-24 saw a cumulative increase of 0.012 and 0.027 respectively in weights. Interestingly, most of the weight of the failed DIP was assigned to DIP-27 to DIP-30 (cumulative increase of 0.066). This is due to better latencies on DIP-29,30 for the same weights. We detect the DIP failure within 100 msec (§4.5). The ILP ran immediately and took around 120 msec to recompute the weights and then program the weights. HAproxy changed weights immediately. Recall that the latency-weight curve calculation is *not in the critical path* – we only need to rerun the ILP using current latency-weight curve (§4.5).

Addressing change in capacity: We reduce the capacity of DIP-25 to DIP-28 by co-running a process that consumes 1 core. Total traffic is unchanged. This change in capacity was reflected in latency differences for the same weights, and we update the weight-latency curve for DIP-25 to DIP-28 (§4.5). Fig.16(b) shows that KnapsackLB is able to react to the capacity change and adjust the weights. Again, instead of reducing the weights of DIP-25 to DIP-28 by 25%, KnapsackLB reduced the weights assigned to these DIPs by 15-17%. The remaining weight was mostly assigned to DIP-29,30, mainly because such DIPs with higher capacity had more room to absorb traffic with respect to the increase in latency. The weights were not equally split as as the ILP made latency-informed decisions. None of the VMs were overloaded.

As we performed latency measurements every 1 second (only failure detection probes run at 100 msec), the above dynamic was detected within 1 second. The ILP took roughly 120 msec to calculate the new weights.

Addressing change in traffic: KnapsackLB also effectively handle changes in traffic as detailed in Appendix (§B). This is similar to handling change in capacity.

6.5 Comparing Against Agent-based Method

We compare KnapsackLB against agent-based method such as Cheetah[19], where we have an agent on each DIP to measure its CPU utilization and adjust the load to get uniform CPU utilization. Note that, as mentioned in §2 and §3, simply measuring CPU utilization falls short in optimizing for performance: (a) as performance depends on many factors including hardware contention (cache, memory bus etc.), throttling at hypervisor and application queues, (b) when instances have different performance (latency) for the same CPU utilization (e.g., DS and F type VMs), (c) such a method is agent-based while KnapsackLB is agent-less. We use 4 DIPs of same VM-type but reduce the capacity of one DIP to 75%. Cheetah computes weights *iteratively* – each iteration computes the weights using current CPU load and terminates when CPU load is roughly same. In contrast, KnapsackLB uses pre-computed weight-latency curve to speedup weight adjustments.. Such an algorithm took 4× time to get weights for uniform CPU utilization compared to KnapsackLB.

6.6 Load Balancing using Other LBs

We demonstrate that KnapsackLB can work with: (a) Nginx[12]: another widely used L4 LB that provides an interface to specify the weights. (b) Azure: LB does not provide an interface to specify weights. We detail it in Appendix (§C).

Table 4. ILP running time across different #DIPs.

#DIPs	10	50	100	500	1000
time (msec)	20	194	645	5.8K	21.1K

6.7 Simulation with Large Number of DIPs

Now we turn to ILP running time and accuracy of multi-step ILP. We use simulations for large number of DIPs. For each DIP, we keep the capacity the same and use the latency-weight curve for the F-series VM from the previous section, and the traffic set to 80% of the total capacity.

ILP running time: We measure the ILP running time (shown in Table 4) as we vary the number of DIPs from 10 to 1000. We feed 10 points uniformly between 0 and w_{max}. It can be seen that ILP runs quite fast: when the number of DIPs is smaller than 100, the ILP completes in 645msec. When the number of DIPs is 1000, the ILP finishes in 21sec.

Multi-step ILP: We now compare the accuracy and running time by using multi-step ILP as described in §4.4. There are 100 DIPs. First, we feed 100 weights uniformly between 0 and w_{max}. Next, we feed in 10 weights in two steps. We choose 10 as a sweet spot between speed and available weight options. We found that multi-step ILP reduces the run-time by 28.3× while sacrificing only 0.1% accuracy (Table 5).

6.8 Overheads at Large Number of DIPs

Lastly, we show that the overheads (in terms of cores and costs) for KLM, latency store, KnapsackLB controller are very miniscule (Appendix §D).

Table 5. Evaluating multi-step ILP. For 10 #points, we run ILP twice.

#points	running time	accuracy
100	36.8sec	100%
10	0.65sec x2	99.9%

7 Related Work

Performance variability and prediction: Performance variation has been acknowledged for many years. Google reported up to 20% capacity reduction due to noisy neighbors[8]. Similarly, [18, 39, 55] report up to 40% change in capacity. [34, 63] report up to 450% degradation in performance due to noisy neighbors in storage. Other works also shed light on performance variability ([20, 24, 25, 30, 41, 43, 56, 58]). Recent works have also focused on predicting the performance due to resource contention (*e.g.,* Ernest[57] for analytic workloads on cloud, Paris[61] for choosing best VMs in cloud). However, they use offline profiling of workloads. In contrast, KnapsackLB is completely online and does not require any profiling apriori. [22, 39, 55] provide performance variation and prediction for NFV, but rely on access to hardware counters (e.g., cache misses) that unfortunately are not available to tenants (or to KnapsackLB) in the cloud. That said, even if such counters become available, adjusting weights based on the counters remains challenging.

LB designs: Recent LB works focus on cost, availability, scalability. Ananta[48] and Maglev[23] are running in production systems. [28, 44, 64] use hardware to save costs. [19, 47] focus on improving the resiliency of LB. KnapsackLB is complementary to these designs and adjusts the weights based on performance. [49, 51, 62] rely on DIP counters or changes to network/MUXes for LB. [59] also works with queues tied to the applications. However, it requires clients sending probes, or servers sharing queue lengths. In KnapsackLB, clients are running inside customer VMs or running on Internet that cannot be changed to send probes. KnapsackLB does not require any changes to clients, servers or MUXes.

Latency as a congestion signal: Prior works have also used latency as a signal for congestion control including Timely[45], Swift[33] and BBR[21]. KnapsackLB uses the latency as congestion signal for load packing.

Capacity based packing: Prior works have looked at capacity based packing but in different context. E.g., multi-resource cluster scheduling [31], VM/container placement [36, 37, 40], CDN routing [26], replica assignment [35, 54]. Such works use intrinsic resource usage, feedback from the servers and/or assistance from clients. E.g., [54] requires servers to send queue sizes and clients to rank servers. [26] requires changes to MUXes, DIPs to gather and dissipate load details and rerouting. KnapsackLB neither requires such information nor control on DIPs, MUXes and clients.

8 Conclusion

We present KnapsackLB to empower other layer-4 LBs to balance load according to the performance of backend instances (DIPs). KnapsackLB is agent-less, fast and versatile – suitable to use with a wide variety of LBs. KnapsackLB builds weight-latency curves, formulates an ILP problem to minimize overall latency, and proposes techniques to expedite the ILP computation. Using a prototype and large-scale simulations, we show KnapsackLB cuts latency by up to 45%.

Acknowledgements

We thank the anonymous reviewers of CoNEXT 2024 for many insightful comments that helped improve the quality of this paper. This work was partly supported by the U.S. National Science Foundation awards 2019302, 1910796, and 2422076, and gifts from Meta, Network Programming Institute (NPI), Xilinx, and the eBPF foundation.

References

[1] Azure L4 LB policy. https://learn.microsoft.com/en-us/azure/load-balancer/concepts.
[2] Azure Redis cache. https://azure.microsoft.com/en-in/products/cache/.
[3] Azure spot VM pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/.
[4] Azure traffic manager. https://learn.microsoft.com/en-us/azure/traffic-manager/.
[5] COIN-OR LP solver. https://www.coin-or.org/.
[6] DS series Azure VMs. https://learn.microsoft.com/en-us/azure/virtual-machines/dv2-dsv2-series.
[7] F series Azure VMs. https://azure.microsoft.com/en-us/blog/f-series-vm-size/.
[8] Google LB insights. https://sre.google/sre-book/load-balancing-datacenter/.
[9] HAProxy LB. https://www.haproxy.org/.
[10] LB DIP selection algorithms. https://www.haproxy.com/solutions/load-balancing/.
[11] Microsoft YARP. Yet Another Reverse Proxy. https://microsoft.github.io/reverse-proxy/.
[12] NGINX load balancer. https://docs.nginx.com/nginx/admin-guide/load-balancer/tcp-udp-load-balancer/.
[13] PuLP Python binding. https://coin-or.github.io/pulp/.
[14] Redis in-memory data store. https://redis.io.
[15] Redis pricing. https://azure.microsoft.com/en-in/pricing/details/cache/.
[16] VMware AVI LB. https://nsx.techzone.vmware.com/resource/vmware-nsx-advanced-load-balancer-avi.
[17] J. a. T. Araújo, L. Saino, L. Buytenhek, and R. Landa. Balancing on the edge: Transport affinity without network state. In *USENIX NSDI 2018*.
[18] A. O. Ayodele, J. Rao, and T. E. Boult. Performance measurement and interference profiling in multi-tenant clouds. In *IEEE International Conference on Cloud Computing 2015*.
[19] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr, P. Papadimitratos, and M. Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *USENIX NSDI 2020*.
[20] Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, and E. Zadok. On the performance variation in modern storage stacks. In *USENIX FAST 2017*.
[21] N. Cardwell, Y. Cheng, S. H. Yeganeh, and V. Jacobson. Bbr congestion control. *Working Draft, IETF Secretariat, Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00*, 2017.
[22] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *USENIX NSDI 2012*.
[23] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX NSDI*, 2016.
[24] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder. Temperature management in data centers: Why some (might) like it hot. In *ACM SIGMETRICS 2012*.
[25] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *ACM SoCC 2012*.
[26] A. Flavel, P. Mani, D. Maltz, N. Holt, J. Liu, Y. Chen, and O. Surmachev. FastRoute: A Scalable Load-Aware Anycast Routing Architecture for Modern CDNs. In *USENIX NSDI*, 2015.
[27] R. Gandhi, Y. C. Hu, C.-k. Koh, H. H. Liu, and M. Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *USENIX ATC*, 2015.
[28] R. Gandhi, H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. In *ACM SIGCOMM*, 2014.
[29] S. I. Gass. *Linear programming: methods and applications*. Courier Corporation, 2003.
[30] S. Ginzburg and M. J. Freedman. Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms. In *International Workshop on Serverless Computing 2020*.
[31] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM 2014*.
[32] H. Kellerer, U. Pferschy, and D. Pisinger. *Introduction to NP-Completeness of Knapsack Problems*. Springer Berlin Heidelberg, 2004.
[33] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM 2020*.
[34] M. Kwon, D. Gouk, C. Lee, B. Kim, J. Hwang, and M. Jung. Dc-store: Eliminating noisy neighbor containers using deterministic i/o performance and resource isolation. In *USENIX FAST 2020*.
[35] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *USENIX OSDI 2020*.
[36] Y. Li, X. Tang, and W. Cai. On dynamic bin packing for resource allocation in the cloud. In *ACM SPAA 2014*.
[37] L. Lu, H. Zhang, E. Smirni, G. Jiang, and K. Yoshihira. Predictive vm consolidation on multiple resources: Beyond load balancing. In *IEEE/ACM IWQoS 2013*.

[38] A. S. Manne. On the job-shop scheduling problem. *Operations research*, 8(2), 1960.

[39] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry. Contention-aware performance prediction for virtualized network functions. In *ACM SIGCOMM 2020*.

[40] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu. Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *IEEE IPCCC 2017*.

[41] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci. Taming performance variability. In *USENIX OSDI 2018*.

[42] S. Martello and P. Toth. *Book, Knapsack Problems: Algorithms and Computer Implementations.* 1990.

[43] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS 2015*.

[44] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[45] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM 2015*.

[46] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[47] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with beamer. In *USENIX NSDI 2018*.

[48] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. In *ACM SIGCOMM*, 2013.

[49] E. Qin, Y. Wang, L. Yuan, and Y. Zhong. Research on nginx dynamic load balancing algorithm. In *International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, 2020.

[50] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM 2015*.

[51] B. V. Shobhana, S. Narayana, and B. Nath. Load balancers need in-band feedback control. In *ACM HotNets 2022*.

[52] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *ACM MICRO 2015*.

[53] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *IEEE HPCA 2013*.

[54] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX NSDI*, 2015.

[55] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. Resq: Enabling slos in network function virtualization. In *USENIX NSDI 2018*.

[56] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup. Is big data performance reproducible in modern cloud networks? In *USENIX NSDI 2020*.

[57] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *USENIX NSDI 2016*.

[58] G. Wang, L. Zhang, and W. Xu. What can we learn from four years of data center hardware failures? In *IEEE DSN 2017*.

[59] B. Wydrowski, R. Kleinberg, S. M. Rumble, and A. Archer. Load is not what you should balance: Introducing prequal. In *Usenix NSDI 2024*.

[60] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li. Dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *ACM EuroSys 2018*.

[61] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *ACM SoCC 2017*.

[62] Z. Yao, Y. Desmouceaux, J.-A. Cordero-Fuertes, M. Townsley, and T. Clausen. Hlb: toward load-aware load balancing. *IEEE/ACM Transactions on Networking*, 2022.

[63] J. Yi, B. Dong, M. Dong, R. Tong, and H. Chen. {MTˆ 2}: Memory bandwidth regulation on hybrid {NVM/DRAM} platforms. In *USENIX FAST 2022*.

[64] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *USENIX NSDI 2022*.

A Calculating Old Flow Completion Time

We should ensure that we correctly measure the latency as a reflection of changing weights. Once we recalculate the weights for latency measurement, we want to program those weights to the LB dataplane and measure the latency. However, we may be unable to measure the latency right away. There are delays due to (a) the LB controller (Fig.6) taking time to program the dataplane, and (b) only new connections adhering to the new weights once the dataplane is reprogrammed (to preserve connection affinity [23, 28, 48]). In particular, the old connections directed to a DIP due to the old weights continue to influence the DIP's latency after the weight change, resulting in a clouded view of the impact of the weight change.

KnapsackLB's approach is to wait till old connections finish. However, since the KnapsackLB controller does not modify the MUXes or DIPs, it does not know whether the old connections are completed. We calculate the time between setting the weight and latency measurement (called *drain time*) by using some extreme settings: for a DIP, we first set the weights high enough that the latency is high (time T_1). Then we set the weight to 0 so that no new connections go to this DIP. We continuously measure the latency until it reaches l_0 (time T_2). We calculate drain time as $T_2 - T_1$. We measure the drain time every 120 mins (configurable).

B Addressing change in traffic in KnapsackLB

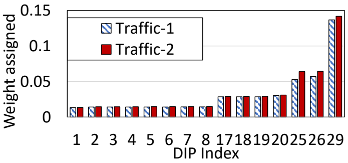


Fig. 17. Weight change due to traffic change.

In this experiment, we have all the DIPs at their original capacity, and we increase the traffic by 10%. We detected traffic increase when all DIPs observe increase in latency for the same weights. In such cases, we shift the weight-latency curve to left. As shown in Fig.17, it can be seen that DIP-25 to DIP-30 absorbed most of the extra traffic. This is again because such DIPs have more room to absorb traffic than other DIPs for the same latency increase. KnapsackLB load balanced the traffic as per capacities without overloading any DIP. The ILP took roughly 120 msec to recompute the weights.

C Load Balancing using Other LBs

Table 6. Fraction of requests received by 3 DIPs using Nginx and Azure traffic manager (TM).

LB	DIP-1	DIP-2	DIP-3
Nginx	20%	30%	50%
Azure TM	18%	34%	48%

We demonstrate that KnapsackLB can work with: (a) Nginx[12]: a widely used L4 LB that provides an interface to specify the weights. (b) Azure: LB does not provide an interface to specify weights. In (b), we use traffic manager (TM) to do load balancing using DNS[4] that resolves the IP based on the weights of the DIPs. In this experiment, we use 3 DIPs behind the above LBs with weights DIP-1 = 0.2, DIP-2 = 0.3, DIP-3 = 0.5. Table 6 shows the fraction of requests received by individual DIPs (total requests = 10K). Nginx does LB as per the weights specified and can work with KnapsackLB similar to HAProxy. For Azure TM, it roughly splits the DNS requests in the weights specified. However, we note that such a load balancing depends on the DNS cache timeout, and clients can see delay in adhering to new weights. This experiment shows that KnapsackLB can work with other LBs, also using DNS when LBs do not provide a native interface to specify weights.

D Overheads at Large Number of DIPs

Table 7. Workload details showing #VIPs for different #DIPs/VIP for a 60K DIP datacenter.

#DIPs/VIP	5	10	50	100	500	1000
#VIPs	2000	1000	200	100	20	10

KLM: KLM sends the latency measurement probes (20 HTTP requests) to individual DIPs every 1 second independent of the DIP size. Additionally, it sends 3 HTTP requests every 100 msec for failure detection. We found the throughput of KLM is 4500 requests/sec on DS1 v2 VM (1 core) in Azure. As the #DIPs/VIP and #VIPs is not publicly available, we use the workload detailed in Table 7 consisting of many mice and a few elephant VIPs, derived from [28]. After considering separate KLM for VIPs ≤225 DIPs, we would need 11K KLM cores for the 60K DIPs from table 7. Assuming DIPs run on D8a type (8 cores; $280/month), and KLM runs on DS1 type (1 core; $41/month), the overhead in terms of #cores and cost is just 2.3% and 2.4% respectively. KLM can run on spot VMs reducing costs by 2.6×[3]. Fig.18 shows the overhead of KLM probes at different intervals. While configurable, we chose probe interval of 1 second as a sweet-spot between overhead and responsiveness.

Latency store: we use Azure Redis cache in the same DC with *Premium* option[15] for good performance. Each Redis *get* operation takes 0.3-4 msec (using persistent connections). For 100K DIPs with 10 latency points per DIP, the total data easily fits within 6GB costing just $6/day (with discount for 3 years subscription), which is a very minuscule overhead.

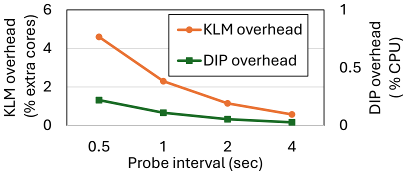


Fig. 18. KLM overhead. The KLM overhead is in terms of number of extra cores normalized to DIP cores. The DIP overhead is CPU utilization to process the KLM requests that is very small (less than 0.5%).

KnapsackLB **controller:** the controller runs: (a) polynomial regression, (b) ILP. For regression, it takes on average 1 msec/DIP on single core. For doing regression for 60K DIPs, it would take 60 cores at the controller. With 8 cores/DIP, the overhead (as number of cores) is just 0.01%. Next, the total running time to run ILP for the workload detailed in Table 7 is 851 seconds on a single 8-core VM. The controller scales easily across VIPs as the VIPs are independent and can run on different VMs running ILPs. Assuming running ILP per VIP every 5 seconds (for dynamics), we would need 193 such VMs (accounting for VIPs that take ≥ 5 seconds to finish ILP). With all DIPs and controller running on 8 core VMs, the overhead (in terms of number of cores and cost) is just 0.32%. Lastly, KnapsackLB can be extended to support larger number of DIPs. In such cases, we can run the ILP in 3 steps (instead of 2 as used in §4.4) to keep the ILP latency bounded.