Beyond Amdahl's Law: Achieving Superlinear Scaling with Distributed Self-adjusting Systems

Paper #619, 12 + 7 pages

Abstract

Conventional wisdom suggests that doubling the number of workers in a distributed system can result in at most 2x performance improvement, but most often less if the system does not provide perfect parallelism. This common wisdom is codified by Amdahl's famous scaling law, asserting sublinear scaling and diminishing returns for parallelization. Curiously, faster-than-linear (superlinear) scaling has also been observed in several real systems, but most reports were dismissed as use-case specific artifacts, elusive interplays between memory and CPU, or mere measurement errors.

In this paper, we argue that superlinear scaling is not just an engineering curiosity, in that distributed systems can be systematically architected to achieve it. Our insight is that dispatching jobs to parallel workers so that the locality of reference in the per-worker input streams increases, and using self-adjusting algorithms so that workers can take advantage of the higher locality to dynamically improve their own performance, together yield superlinear scaling. Using various load balancing policies and self-adjusting algorithms from the literature, we report 100–3000x speedup when scaling distributed list lookup and tree search workloads to 48 CPU cores, up to 70x beyond what is predicted by Amdahl's law. By implementing an earlier self-adjusting packet classifier algorithm in the Linux kernel and combining it with a hash-based load balancer, we obtain 800x speedup for synthetic and 220x speedup for realistic firewall traces on 32 cores, resulting 5–25x times raw performance improvement compared to default Linux packet classifier.

1 Introduction

With the end of Moore's law, computing power in modern systems increasingly comes in the form of parallel processing resources. A major obstacle faced by network engineers is how to harness this increasingly parallel computing power for scaling distributed systems [46, 69, 75, 82, 88].

In horizontally scaled applications, a load balancer dispatches jobs across a fleet of workers that process the jobs in parallel [17]. In the context of *web applications*, HTTP load balancers [12, 21, 64] distribute requests across a swarm of backend web servers. Multicore *OS network stacks* [11, 47, 86] run multiple instances of the networking logic on different CPUs and leverage the NIC to dispatch packets to CPU cores. In massive-scale *key-value stores* [27], the key-space is hashed into multiple shards (partitions) and each shard is assigned to a separate server for processing. The system's overall goal is to achieve the greatest possible parallel speedup with a given number of workers, in order to minimize the execution time of a single task or maximize the number of completed tasks in a given time period.

Suppose a web app handles 100 requests per second using a single server. As we add another server, we expect a throughput of 200 requests per second. In reality, however, we usually obtain slightly less, and this is worsened as the system is scaled up further. This is because some fraction of most workloads is inherently sequential and, therefore, bound to execute on a single CPU core. For instance, the web servers may need to synchronize on a mutex to access global state, which makes all state updates sequential. Beyond a certain threshold parallel performance plateaus as the sequential workload becomes a bottleneck.

The maximum speedup, measured as the ratio of the wall clock times of sequential and parallel execution, is formally described by Amdahl's law [3]. In general, the greater the sequential portion compared to the parallelizable fraction of the code, the more performance is lost compared to an "ideal" linear scaling, and the faster the system reaches saturation (see Fig. 1). Amdahl's law has remained one of the most useful tools in the system engineering toolbox throughout the almost 60 years since its first publication [13, 18, 33, 38, 43, 50, 58, 66].

Inherent to Amdahl's law is that no system can scale faster than linear: doubling parallel resources will yield at most two times the performance. Curiously, there have been several reports on faster-than-linear (*superlinear*) scaling experimentally observed in, e.g., database systems [24, 39], distributed caching [76, 84], SDN analytics [77], high-performance computing [36, 37, 72], multi-robot systems [40], parallel search in information retrieval systems [83, 84], and large-scale network simulations [9] (see full taxonomies in [41, 72]). Many authors argue, however, that superlinear scaling is merely

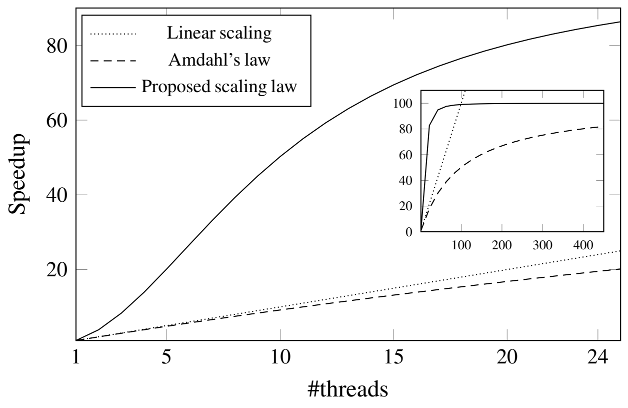


Figure 1: Linear scaling, Amdahl's law and the proposed superlinear scaling, for $s = 0.01$, where s is the fraction of the execution time inevitably spent in the sequential part of the code. Our proposed scaling law suggests that superlinear scaling is feasible for specific workloads accompanied by a right combination of a load balancer and a self-adjusting data structure. The inset shows the asymptotics depicting the saturation of the superlinear scaling of the system.

a byproduct of running memory-/cache-bound applications on a "bigger machine" [41], others are concerned that it is hard to generalize beyond a specific set use cases [41, 72], and some outright dismiss faster-than-linear scaling all together [23, 31], concluding that *"superlinearity, although alluring, is as illusory as perpetual motion"* [32].

In this paper we challenge this view: we show that distributed systems can be methodologically designed for reaching superlinear scaling. Our motivation is that networking applications are often embarrassingly parallel, which may admit a massive superlinear initial growth phase before scaling eventually and unavoidably blocking on a serial bottleneck.

Our main observation is that, to achieve superlinearity, one has to carefully combine an appropriate load balancing policy with a proper worker implementation. Indeed, load balancing in most distributed systems is deliberately designed to improve the locality of reference in the input of the workers: web apps apply the "sticky sessions" rule to route all requests of a particular user to the same web server; networking code often uses IP 5-tuple hashing on the NIC to ensure that all packets of a flow are processed on the same CPU; and key-hashing in sharded key-value stores directs all client queries to a key to the same replica. Each of these load balancing policies tend to make the input stream processed by the parallel workers more predictable, compared to the input processed by the system. Such a *locality boosting load balancer*, paired with a *self-adjusting algorithm* so that workers can take advantage of the higher input predictability to adaptively improve their performance, will, as we show both theoretically and empirically, yield faster-than-linear speedup in a broad range of applications (see Fig. 1).

After some background on Amdahl's law (§2), we present our *distributed self-adjusting system architecture* and show that superlinear speedup is a natural product of combining locality-boosting load balancing with self-adjusting algorithms (§3). Using common list and tree search algorithms from the literature, we achieve 100–3300× speedup in simulations, orders of magnitude surpassing Amdahl's law or even plain, linear scaling. As an unexpected byproduct, we attain linear scaling even when we limit the system to a single CPU core. Then we extend our analysis to real systems (§4): we carefully apply our methodology to engineer a Linux-kernel based packet classifier to reach superlinear scaling. On synthetic and real-life firewall traces, our implementation shows up to 800× faster than linear scaling, 5–25× improvement beyond the default Linux firewall implementation which scales according to Amdahl's law. Finally, we review related work (§5) and draw the conclusions (§6). We note that all code will be available as open source after publication.

2 Background

First we review Amdahl's scaling law and then we show a typical pattern that seems to defy it: distributed caching. In the following, we will use the terms "distributed" and "parallel" interchangeably to connote that a system is scaled to run on multiple independent compute threads ("workers"), e.g., distributed to separate nodes, scaled to parallel CPUs located inside the same node, run in multiple datacenters, etc.

2.1 Amdahl's law

A cornerstone result in parallel computing, Amdahl's law [3] establishes a firm limit on the performance gain one can obtain by distributing a computation task over multiple processors. Given a partially parallel program, denote the fraction of execution time spent in the sequential part of the code by s, and the parallel fraction by $(1-s)$. Here, some code is "sequential" if it cannot benefit from the improvement of the parallel computing resources, like single-threaded code, critical sections guarded by exclusion locks, etc. Denote by $T(k)$ the runtime (in seconds) of the program when executed on k processors, and let $S(k) = \frac{T(1)}{T(k)}$ denote the performance improvement relative to a single-threaded execution (i.e., the *speedup*). Then, the following relation holds (see Fig. 1):

$$S(k) = \frac{T(1)}{T(k)} = \frac{1}{s + \frac{1-s}{k}} \quad . \tag{1}$$

Here, the term $\frac{1-s}{k}$ establishes that the perfectly parallel part of the program executes k times faster on k processors than on a single core. By Amdahl's law, *(i)* no code can scale faster than linear (i.e., $\frac{dS(k)}{dk} \leq 1$, with equality exactly when $s=0$), *(ii)* throwing additional workers on a computation task yields diminishing returns ($\frac{dS(k)}{dk}$ is monotonically decreasing in k) and *(iii)* the asymptotics is limited by the sequential part only ($\lim_{k\to\infty} S(k) = \frac{1}{s}$).

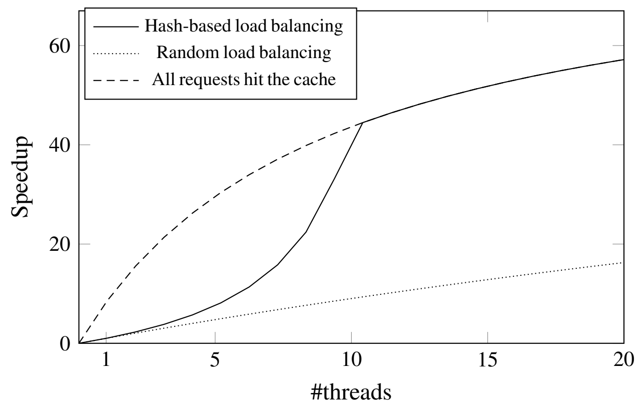


Figure 2: Scaling laws for distributed caching: hash-based load balancing, lower envelope (round robin load balancing) and upper envelope (perfect cache hit rate with k caches).

2.2 Distributed caching defies Amdahl's law

Critical to Amdahl's law is the assumption that the size of the individual sub-problems assigned to the workers remains constant as we scale the system (see also "Gustafson's law" [38]). Under this "fixed size" assumption [37], faster-than-linear scaling is impossible [23]. However, when this assumption fails, say, when the per-worker problem size gets progressively smaller or execution gets gradually faster as we add more parallel workers ("scaled size" model [37]), superlinear scaling is sometimes observed [36, 39, 40, 77, 83, 84]. Note that in the majority of the literature any function growing faster than $f(x) = x$ is considered "superlinear", despite that, e.g., $f(x) = 3x$ is, mathematically, linear. Some authors distinguish these functions using the term "superunitary" [41]. In line with the literature we will use the former terminology below.

The most prominent example for the scaled size model is *distributed caching* [39, 77, 84] (for complete taxonomies see [37, 41, 72]). Most modern CPUs come with unshared Level-1 fast cache memory: the more CPU cores, the more fast memory is available for caching, which improves the cache-hit rate at the workers. This tends to speed up memory/cache-bound code disproportionately. Many distributed applications also contain a fast-path/cache; e.g., `memcached` is often used as a fast cache for a "slow" web service [24, 63], popular keys are cached in the OS kernel for fast key-value store access [27, 54], FIB caches maintain the most recent IP routes to sidestep longest prefix matching [73], hierarchical flow caches serve as a fast-path in programmable software switches [71], etc. All these workloads may benefit from the caches becoming more efficient as the system is scaled and, potentially, show superlinear speedup on certain workloads.

It is instructive to quantify superlinear speedup in this context using a simple model. Suppose a source emits uniformly distributed random requests for m items and requests are distributed among k workers, each using a separate cache of size c, by hashing on the request id. Initially, the cache hit rate for a single worker that processes all m possible requests is $\delta := c/m$. Adding k workers effectively partitions the requests into k random buckets so that each worker will perceive uniformly distributed requests for only m/k items, which improves the cache hit rate at each worker to $\frac{c}{m/k} = k\delta$ $(k\delta \le 1)$. This puts the lookup time of the system of k parallel caches to

$$T_c(k) = \begin{cases} s + \frac{1-s}{k}(k\delta + (1-k\delta)\rho) & \text{if } k\delta \le 1 \\ s + \frac{(1-s)}{k} & \text{otherwise} \end{cases}, \qquad (2)$$

where ρ is the penalty for a cache miss event, δ is the cache hit rate for a single worker, and s denotes the fraction of execution time spent in the sequential part of the code.

The speedup $S_c(k) = \frac{T_c(1)}{T_c(k)}$ for the parameters $s = 0.1, \delta = 0.1$ and $\rho = 10$ is depicted in Fig. 2. The lower envelope of the scaling profile is given by Amdahl's law for the system with random or round robin load-balancing. As k grows the scaling profile progresses over a superlinear curve to an elevated Amdahl's law profile, representative of a system serving *all* requests from fast memory. Note that this occurs *only* if request dispatching is chosen carefully to partition the item space. Modulo hashing assigns the same item to the same worker deterministically, so that workers process only a subset of the items that may have a greater chance to fit into the cache. In contrast, a random or a round robin load balancer may assign any item to any worker, which defeats the purpose of improving workers' cache hit rate.

Currently the only general methodology to achieve faster-than-linear scaling seems to require deploying additional fast caches. Moving an application to a "bigger machine" [84], however, is not always feasible due to, e.g., physical or financial constraints. In some cases caching cannot be used at all (e.g., for inherently stateful computations or complex database queries) or it introduces more overhead than it saves (e.g., for predominantly uniform input or rapidly changing data). Moreover, caching comes with certain extra complexity and often cache invalidation and eviction policies and data consistency mechanisms are too costly to implement in a massively distributed setting [76]. Apart from caching, however, currently the only way to achieve faster-than-linear scaling is to rely on piecemeal problem-specific techniques, comprehensive domain knowledge, and pure luck [41, 72]. And even then, some compellingly argue that superlinear growth itself is a performance illusion, which goes against the very laws of nature much like perpetual motion [31, 32]

3 Beyond Amdahl's law: Distributed self-adjusting systems

We now present a general distributed systems architecture, which, as we show theoretically and empirically later, produces faster-than-linear scaling in several disparate problem domains. Our main observation is that, whenever genuinely observed, superlinear scaling assumes two critical components: a policy to dispatch jobs to workers in a way to increase the locality of reference in the per-worker input streams, plus an algorithm

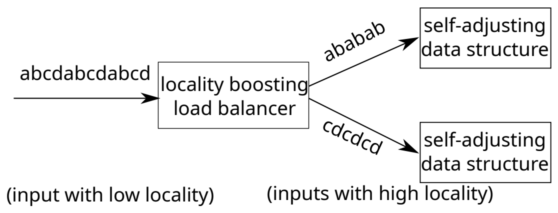

abcdabcdabcd
locality boosting load balancer
ababab
cdcdcd
self-adjusting data structure
self-adjusting data structure
(input with low locality)
(inputs with high locality)

Figure 3: A locality boosting load balancer partitions the input sequence of a given locality into subsequences with higher locality. Self-adjusting data structures perform better on inputs with higher locality.

that can adaptively exploit the structure in the input to process it more efficiently. Our architecture is purely a software technique in that it does not require the addition of new cache space to a system. Nonetheless, it contains distributed caching as a special case and hence automatically takes advantage of additional fast memory, if available.

3.1 Locality-boosting load balancing

The first crucial component in our architecture is a locality-boosting load balancer. In this context, load balancing refers to the distribution of computational work or incoming network traffic across multiple parallel *workers* (servers, processors, or nodes). A good load balancing strategy ensures optimal resource utilization, minimizes response time, avoids overloading any single resource, and, as we argue below, improves the locality in the input presented to the workers.

In the context of this paper, *locality of reference* is the property of a sequence of inputs that subsequent items are statistically dependent on each other. Such structure in the input can then be readily exploited by the proper algorithm [8, 14, 15, 42, 42, 70, 81] or a runtime optimization framework [26, 52, 55, 61] to improve the performance of the code that processes it. A request set with minimal locality is uniformly distributed on the entire input domain and hence unpredictable, while one with maximal locality contains only a single item, i.e., maximally predictable. A *locality-boosting load balancing* policy is then a request dispatching strategy that can statistically or deterministically improve the locality of reference experienced by the worker threads, *turning an unpredictable system input into multiple streams of predictable input* to be processed by the workers (see Fig. 3).

We distinguish two types of locality in this context. *Spatial locality* means that the distribution of requests on the input domain is statistically biased towards a particular subset of the items. One way to ensure this in the load balancer is to *partition* the input domain into disjoint subsets, so that worker's input distributions are concentrated on a smaller set of inputs. For instance, the hash-based load balancer we used previously to show superlinear scaling with distributed caching is such a partitioning load balancer. In contrast, a round robin or a uniform random load balancer will export its own spatial input locality unchanged to the workers. A related concept is *temporal locality*, which refers to the reuse of specific items in the input within a relatively small time duration. One way to boost temporal locality is to reorder items within a time window: e.g., Reframer applies controlled delays to particular packets in a packet batch to boost temporal locality and, thereby, enable more efficient processing [26, 52].

3.2 Self-adjusting algorithms

The second critical enabler for superlinear scaling is *self-adjusting algorithms*. Self-adjustment is a general term referring to the property of a dynamic data structure to *automatically reorganize itself based on the sequence of inputs it receives*, in order to optimize performance for future operations on frequently accessed items. Internal data reorganization always introduces extra complexity and overhead compared to a static data structure. Thus, self-adjustment can improve performance only if the input processed by the algorithm exhibits a certain amount of spatial or temporal locality.

Next we review the most prominent self-adjusting data structures from the literature (but see also [8, 16, 70]).

Caches. As the simplest but most universal self-adjusting data structure, caches can serve frequently accessed items fast by storing them in a software or a hardware fast memory. This is typically much faster than if we had to run the request through the full processing pipeline or the slow backing store. Caches have that almost magical capability of self-adaptation, without us having to engineer any prior knowledge of the input into the cache mechanism apart from a promise that it has nontrivial locality. When the promise is true, caches are an inexpensive way to improve throughput and response time. When there is no locality in the input, however, caches usually just add extra latency and overhead. Note that caches do not necessarily have to come in the form of hardware memory: a fast key-value store is a candidate cache for a slow database [24], a kernel fast-path flow cache is a useful way to speed up a slow user-space software switch [71], etc.

List lookup. One of the most widely used self-adjusting data structures is the *move-to-front list*. Suppose we wish to store a list of m items in a way so that reordering, insertion and deletion are fast, while lookup is also reasonably efficient. A straightforward choice is a static linked list. Here, the cost of accessing an item at position i is exactly i. Then, any linked list can easily be upgraded to a self-adjusting list using the move-to-front (MTF) heuristics: after accessing an item it is moved to the front, which improves lookup time for future requests of the same item at minimal cost (see Fig. 4). The MTF heuristics comes with appealing theoretical properties, namely that blindly moving the accessed item to the front of the list is close to the best reordering strategy one could design, even if one knew all future requests [80]. MTF lists handle both spatial and temporal locality. For uniformly distributed input MTF lists usually add nontrivial overhead compared to static lists due to the frequent and useless relinking of the list.

Classic applications of MTF lists are information retrieval systems, compression [15], etc. In general, any use case is a

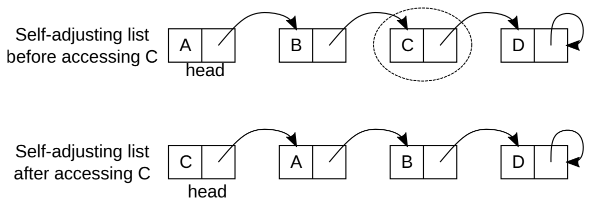


Figure 4: A self-adjusting list containing nodes A, B, C and D serves the request to C and moves C to the front of the list to speed up future accesses to C.

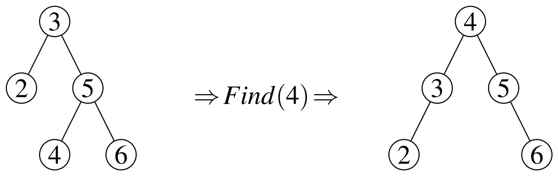


Figure 5: Splay-tree with elements 2, 3, 4, 5, 6. After accessing node 4 it is moved to the root that makes a subsequent lookup to the same node faster, while the tree is well-balanced.

potential candidate application for MTF where the task is to match a request against a list of complex rules that do not lend themselves readily to be arranged into a fast lookup structure (e.g., a search tree), like inference in explainable rule-based AI [20], rule matching in OpenFlow and P4 reference software switches [65], packet classification in networking (see later), etc. We note that caching is a subset of list lookup, in that every algorithm for list reorganization gives rise to a different cache management algorithm [80].

Search trees. A search tree is an efficient tree data structure for locating specific keys from within an ordered set. A *splay tree* is a self-adjusting version of a static search tree, in that it can dynamically reorganize itself by moving popular items closer to the root of tree and less frequently accessed elements to the bottom, while keeping the tree relatively well-balanced [8, 16, 81]. Since access time in a search tree is determined by the depth at which the requested item is to be found, splay trees can improve future access to the same or similar items when the input exhibits temporal or spatial locality (see Fig. 5). Note that a red-black tree, an AVL tree or any similar self-balancing tree is not self-adjusting, in that it can rearrange only with respect to the items *stored* in it but not with respect to the queries *posed* to it.

Splay trees are widely used to adaptively speed up associative memory and data compression algorithms [45], as well as a building block for more complex self-adjusting algorithms.

3.3 Superlinear scaling

So how can locality-boosting load balancing and self-adjusting algorithms, when used together in a distributed system, produce superlinear scaling? First, we present a demonstration on a particular instantiation of the architecture, *distributed self-adjusting list lookup*, and then we provide a formal scaling characterization for general distributed self-adjusting systems.

Consider a partitioning load-balancer (see Fig. 3) combined with a self-adjusting move-to-front list (see Fig. 4) implemented in the workers. Suppose that there are m items to be stored in the list and k workers, each maintaining an independent index into the list. To make things more complicated, we assume uniform request distribution on the entire input domain m at the system's input, which is, recall, the worst case for any self-adjusting algorithm by being totally *unpredictable*. Thus, for a single worker move-to-front reordering has no useful effect and the worst case access time is m, identical to that of a static linked list.

Now suppose we move from 1 worker to k parallel workers where $k \leq m$. This results, within our architecture, that the load balancer effectively partitions the uniformly distributed input on m items into k uniformly distributed input streams for only $^m/_k$ different items (see Fig. 3). This means that the workers' input features a higher spatial locality than the system's input (which sports none). Had we used a random or a round robin load balancer the workers would still see all the m possible inputs, just with a sampled uniform distribution, and no locality. After a while, each MTF list in the workers will have its specific subset of $^m/_k$ items moved to the first $^m/_k$ positions (in an arbitrary order), reducing the worst-case lookup time from m (1 worker) to $^m/_k$ (k workers). This introduces $k\times$ speedup compared to the single-threaded case.

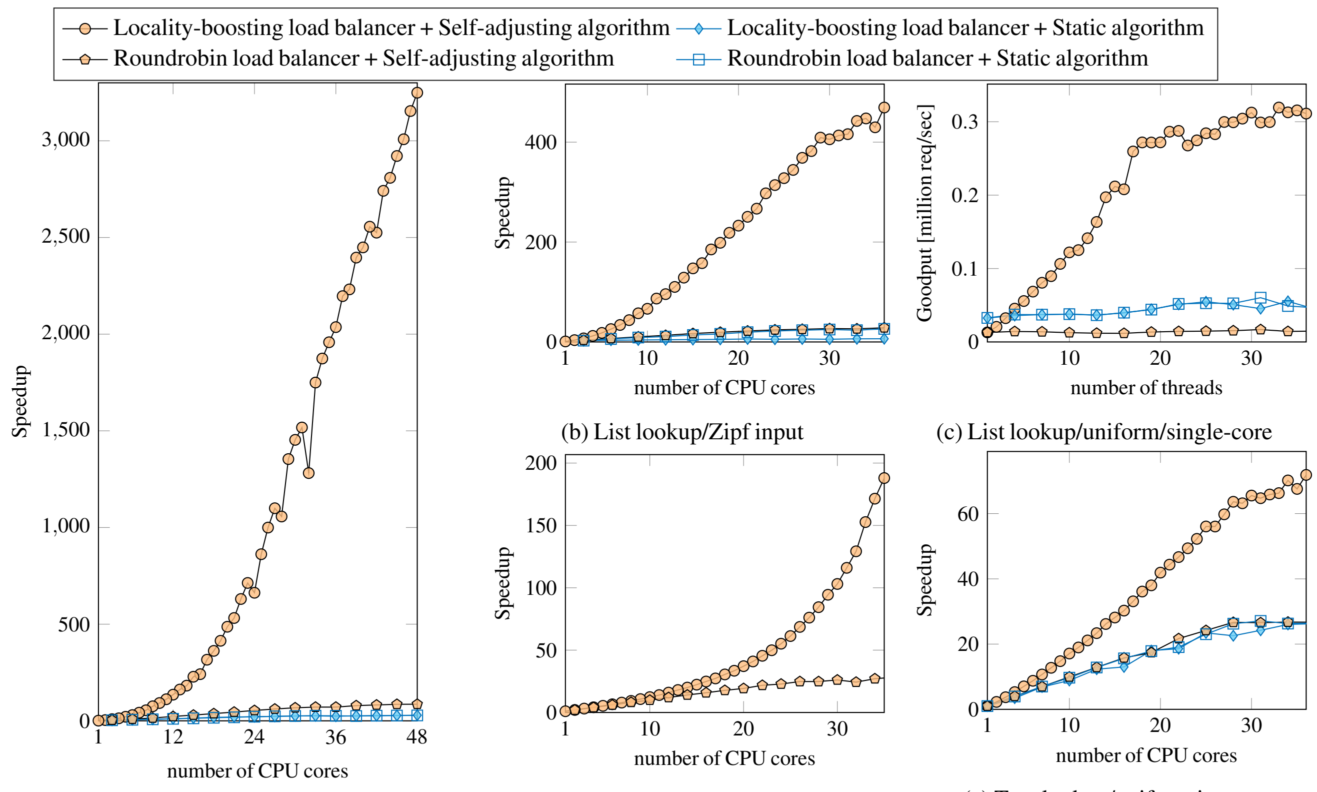
Then, superlinear speedup is merely a product of two simultaneous $k\times$ speedup factors: one $k\times$ factor comes from the self-adjusting list getting progressively faster as we add new workers (recall the "scaled size" model from §2.2), and another $k\times$ speedup because we scale the total compute capacity available to the system k times. The effective speedup is then just the multiple of the two, yielding k^2 times speedup in total. Plugging into Amdahl's law we get the *scaling law for distributed MTF lists on uniform input* (see Fig. 1):

$$S_l(k) = \frac{T_l(1)}{T_l(k)} = \frac{1}{s + \frac{1-s}{k^2}} \qquad k \leq m \ , \tag{3}$$

where s denotes the fraction of execution time spent in the sequential part of the code.

For small values of k, we obtain $O(k^2)$ scaling. As k grows sufficiently large, say, when $k = m$, the workers' input reduces to a singleton ($^m/_k = 1$). From this point the distributed MTF list reduces into a simple parallel hash table and superlinear speedup degrades into an "ordinary" Amdahl's scaling profile until speedup eventually blocks on a serial bottleneck (e.g., the sequential load balancer). For anything between, the system adaptively finds the best combination of an MTF list and a hash-table, producing a quadratic scaling.

In general, superlinear speedup emerges as the superposition of two related speedup factors. First, by splitting the input into multiple input streams of improved locality, the locality-boosting load balancer reduces the "effective size" of the workload the workers will have to process. Denote by $\ell(k)$ the "workload reduction" attainable on the given input this way with k workers. Second. there is a "parallelizability" gain, de-



(a) List lookup/uniform input

(b) List lookup/Zipf input

(c) List lookup/uniform/single-core

(d) Cache lookup/uniform input

(e) Tree lookup/uniform input

Figure 6: Static vs. self-adjusting distributed systems scaling laws with round-robin and hash-based load balancing: (a) static vs. MTF list access speedup on uniform input (m=100k); (b) static vs. MTF list speedup on skewed input (m=100k, Zipf power law with $\alpha = 1.01$), (c) static vs. MTF list access goodput with multiple threads running on a *single core* for uniform input (m=10k); (d) cache access on uniform input (m=50k, $\delta = 0.05$, $\rho = 100k$ cycles); and (e) static balanced vs. splay tree speedup ($m = 500$, $w = 100k$ cycles). Panels (a), (b), (d) and (e) show multicore speedup as the function of the number of CPU cores, each running a single worker, while (c) shows the single-core throughput (goodput) using an increasing number of lightweight parallel threads.

note this by $q(k)$, that is obtained by the self-adjusting workers processing the reduced workloads. In the Appendix we present a formal definition of these terms and characterize the below general *scaling law for distributed self-adjusting systems*:

$$S(k) = \frac{T(1)}{T(k)} = \frac{1}{s + \frac{1-s}{q(k)\cdot\ell(k)}} \quad .$$

If $q(k)\cdot\ell(k) > k$, then we say we achieve superlinear scaling.

Our formal characterization in the Appendix describes the cases when superlinear scaling may be attainable (positive results) and when it cannot (negative results). In general, the attainment of superlinear scaling for some k depends on whether we have the right combination of the input stream and the load balancer. However, even if an input sequence can scale superlinearly for certain values of k, it cannot sustain superlinear scaling for all k. Eventually, superlinear growth peters out and the system falls back to sublinear scaling. Nonetheless, even if superlinear is only an ephemeral phenomenon, it can bring a significant performance boost.

3.4 Empirical evidence for superlinear scaling

Next, we present a series of simulation studies to confirm that locality-boosting load balancing combined with self-adjusting workers (but only this combination) yields faster-than-linear scaling over a broad selection of load balancing policies, self-adjusting algorithms, and input distributions.

Our simulator was written in Go, using lightweight threads (goroutines) managed by the Go runtime to run a given number of workers in parallel, a home-grown implementation of static and MTF lists, and standard Go modules for LRU caches [30], static balanced trees [28] and splay trees [29]. In order to make tree lookup CPU bounded we used an "expensive" order underneath the tree, where every comparison operation costs a configurable w number of extra cycles. The simulator creates the specified combination of a load balancer, k worker threads running the selected lookup algorithm, and a random input sequence with a given request distribution, and then performs a configurable number of lookup operations and measures the total execution time with nanosecond precision. To obtain a full picture, the total execution time includes the transient time needed to warm up the self-adjusting algorithms as well as the overhead of request generation, goroutine scheduling, and memory management. For the specification of the evaluation platform, refer to §4.3.

Fig. 6 shows the results. First, the immediate observation is that *the right combination of a locality-boosting load balancer and a self-adjusting algorithm robustly delivers superlinear speedup*, irrespectively of the problem domain or the input distribution. Even for a worst-case uniform input we obtain 3,300× speedup for list access on 48 CPU cores, almost 70× of "ideal" linear speedup, 200× speedup on LRU caches and 65× speedup on tree search with 36 CPU cores. In most cases, the superlinear growth is so dominant that we can hardly put Amdahl's scaling on the same diagram. Second, *only the combination of locality-boosting load balancing and self-adjusting algorithms produces superlinear speedup*, all other combinations (i.e., round robin with any algorithm or static algorithm with any load balancer) fall back to Amdahl's scaling. Third, *self-adjustment clearly has its overhead*. This can be observed in Fig. 6c that shows the absolute throughput instead of the relative speedup. Here, the single-threaded self-adjusting algorithm is slower than the static one due to processing an unpredictable input. Fourth, *the overhead of self-adjustment is irrelevant for more than one worker, or with skewed request distributions*. For instance, on a Zipf input distribution (Fig. 6b) even the single-threaded self-adjusting version is already 2–2.5× faster in an absolute term (not shown in the figure). However, *only* combined with a locality-boosting load balancer it produces superlinear speedup.

Finally, we show that our architecture can yield a linear parallel performance gain even if we do not even add additional CPU to the system. Fig. 6c shows an evaluation that was executed with an *increasing number of parallel threads sharing a single CPU core*, with a little surplus CPU for the load balancer. The results indicate that the parallel self-adjusting system (but *only* this combination!) delivers linear speedup. How can parallelization benefit performance when the total CPU power available to the system is kept constant? Recall, in the multicore case superlinear speedup emerges due to the superposition of two independent $k\times$ speedup trends, one delivered by self-adjustment and another by the $k\times$ scaling of the total CPU power. When the total available CPU is limited only first $k\times$ speedup factor is in effect, resulting in the observed linear scaling trend.

It is important to stress that superlinear speedup is only possible with respect to a single-threaded single-core baseline. Had we normalized with respect to a multi-threaded baseline constrained to a single core (as in Fig. 6c) we would see only a linear speedup. See the Appendix for a precise formal characterization of the conditions under which superlinear speedup can be attained.

4 Case study: Achieving superlinear scaling in the Linux kernel

We present a case study for systematically applying the distributed self-adjusting systems architecture to a common networking problem: software packet classification [35]. The goal is to demonstrate the general engineering methodology by assembling *existing* techniques into a distributed self-adjusting scheme and understand when, and to what extent, superlinear scaling emerges. We consider it a success if we can robustly reproduce faster-than-linear growth on some realistic workloads. It is a stated *nongoal* to conceive novel algorithms, let alone produce the fastest software packet classifier. Yet, our self-adjusting firewall implemented in the Linux kernel will prove several times faster than the default Linux kernel implementation on a wide range of workloads.

To achieve superlinear scaling we need a self-adjusting algorithm in the first place (plus a locality-boosting load balancer). From the many potential use cases [8, 14, 15, 42, 42, 70, 81] we eventually chose packet classification for the following reasons. First, the default Linux firewall implementation, `nftables`, uses a static doubly linked list to evaluate classifier rules, which makes it an appealing candidate for applying the move-to-front (MTF) heuristics (but see ramifications related to handling rule-dependencies below). Second, underlying packet classification, there is an infamously difficult theoretical problem [35, 49, 49, 67, 68, 79, 87], and achieving superlinear speedup on such a hard problem promises massive performance gain. Third, the Linux kernel network stack offers several flexible software and hardware based load balancers for dispatching packets to parallel classifier instances running on different CPU cores [74], which we will reuse to implement the locality-boosting load balancer component. And fourth, packet classifiers are very difficult to cache [19] (recall, caches are the "cheap" way to obtain superlinear scaling), which calls for a true self-adjusting packet classifier.

4.1 Self-adjusting packet classification

A network firewall is a means to control incoming and outgoing network traffic based on user-defined packet classifier rules (see Fig. 7). A classifier *rule* is a pair of a filter, a user-defined regular expression defined on specific fields of the packet header or metadata, and an action that decides what to do with the packets that match the filter (accept, drop, log, etc.). Rules are organized into linear chains ordered by rule priority. When a packet enters a chain, it is compared against the first rule. If there is a match, the corresponding action is executed and the lookup is over. Otherwise, subsequent rules are matched in priority order until the first match is found.

The `nftables` kernel engine adds a virtual machine to the Linux kernel that uses a DSL for parsing and matching packet header fields [62]. This makes `nftables` agnostic to specific network protocols, in contrast to, e.g., `iptables`, which contains an embedded protocol parser. Currently, `nftables` is the default packet classifier in most Linux distributions.

One way to make `nftables` self-adjusting would be to replace the static linked list it uses internally for rule matching with a self-adjusting list. A naive application of MTF, however, would easily break the semantics of the firewall. This is

Prio	Proto	Src IP	Dst IP	Dst Port	Action
1	UDP	192.168.178.33	23.0.0.45	53	ACCEPT
2	TCP	10.10.10.0/24	23.0.0.45	443	DROP
3	UDP	192.168.178.0/24	23.0.0.45	53	DROP
4	TCP	10.10.10.10/32	23.0.0.45	ANY	ACCEPT
5	IP	192.168.0.0/16	23.0.0.0/8		ACCEPT

Figure 7: Sample firewall rule set. Source ports do not matter.

because rules in the chain may not be independent from each other, and hence may not be freely swapped [49].

Consider the example in Fig. 7 and suppose that, initially, rules are ordered priority-wise in the list: $\langle 1,2,3,4,5\rangle$. Suppose that a packet with the IP 5-tuple (192.168.0.1, 23.0.0.45, UDP, 1, 3478) enters the classifier, where the fields in the 5-tuple are IP source and destination address, protocol, and source and destination port, respectively. Rules are inspected in linear order until rule 5 is found as the first match, at which point the lookup terminates with the verdict ACCEPT. Now, a naive application of MTF would move rule 5 to the front of list, resulting in the order $\langle 5,1,2,3,4\rangle$. Suppose another packet with the 5-tuple (192.168.178.1, 23.0.0.45, UDP, 1, 53) is to be processed next: this will immediately match rule 5 at the front of the list yielding the verdict ACCEPT, despite that, if matched in priority order, rule 3 would be the correct match and the verdict should be DROP.

We say that rule u is *dependent* on another rule v if they have overlapping match criteria in all fields, v has a higher priority than u, and u and v define different actions. Such a dependency means that u is not allowed to be moved before v in the list, otherwise some packets may be erroneously classified. For instance, in the example of Table 7 rule 5 is dependent on rule 3, which is in turn dependent on rule 1, implying the dependency chain $5 \to 3 \to 1$. Similarly, rule 4 is dependent on rule 2.

A dependency-aware variant of the MTF heuristics, called the *Move-recursively-Forward* (MRF) algorithm, is defined in [1] (see Alg. 1). The idea is to push an accessed item forward in the list until the first dependency is reached. To prevent the item from blocking behind its direct dependency, the dependency is also moved forward until the first transitive dependency is hit. This process repeats until the head of the list is reached. Independent rules are however free to be moved without restrictions, to the point that if there are no dependencies then MRF simplifies into a plain MTF policy. Contrarily, if the entire rule set is a single dependency chain then no reordering is allowed and MRF degrades into a static list. In general, MRF moves frequently hit rules, with all their dependencies, to the first positions of the chain, which tends to improve lookup performance on high-locality input without jeopardizing the semantics of the classifier [1]. In addition, MRF is "almost"optimal in the same competitive sense as MTF, in that the best reordering one could obtain even if one knew the entire lookup sequence in advance would yield only a small constant factor improvement over MRF.

Going back to our earlier example, after rule 5 is hit in the list $\langle 1,2,3,4,5\rangle$ MRF moves it immediately after the direct dependency 3 along the dependency chain $5 \to 3 \to 1$, 3 is moved to the position after 1, and the recursion ends resulting the order $\langle 1,3,2,5,4\rangle$. If 5 was hit again, the lookup time would be only 4 instead of 5. Then, 5 would be moved forward again, yielding the order $\langle 1,3,5,2,4\rangle$ and a lookup time of 3. Note that dependency chains can be moved by MRF independently from each other: e.g., if 4 was hit first then we would obtain $\langle 2,1,4,3,5\rangle$ in the first iteration and eventually $\langle 2,4,1,3,5\rangle$, with lookup time for 4 dropping from 4 to 2.

Algorithm 1 Move Recursively Forward (MRF)

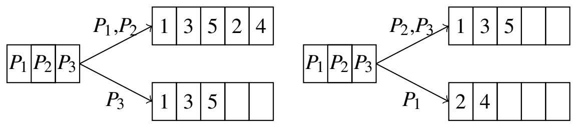
```
1: procedure MRF($y$)
2:     if $y$ has no dependencies then
3:         Move $y$ to the front of the list
4:     else
5:         Let $z$ be the direct dependency of $y$
6:         Move node $y$ to position$(z)+1$
7:         MRF($z$)
8:     end if
9: end procedure
```

We created a comprehensive self-adjusting packet classifier implementation on top of `nftables` using the dependency-aware MRF algorithm [1]. Our implementation can run multiple MRF instances in parallel, each maintaining its own local rule order in a private per-CPU array of pointers that index into a shared static rule list. Apart from lockless list reordering, this also enables lockless rule addition/deletion: every time the rule list is updated we simply allocate a new pointer array at each CPU and update the list head atomically.

The original MRF algorithm uses recursion (see Alg. 1), which may be expensive in the Linux kernel due to the overhead of maintaining the function call stack. To avoid this overhead, we defined an iterative version of the algorithm. When a rule is to be moved forward, we first check whether it can be swapped with the preceding rule. This is done by checking whether the two rules overlap using a range-based representation, which we extract from the rule's bytecode in the `nftables` virtual machine. If there is an overlap then the rule cannot be moved forward so we restart the process, this time trying to move the blocking dependency forward. Otherwise, the two rules are independent so they are immediately swapped and the iteration moves to the subsequent preceding rule. Reordering terminates when we reach the first position. A more efficient implementation would be to precompute dependencies on rule insertion/deletion and run the MRF algorithm using the cached dependencies; implementing this optimization is for further study.

4.2 Locality-boosting load balancing for packet classification

The other ingredient that we need to achieve faster-than-linear scaling is a locality-boosting load balancer. An ideal load balancer would partition the rule set into disjoint per-worker subsets. This would minimize the size of the *active rule set*



(a) Imperfect separation (b) Perfect separation

Figure 8: Locality-boosting load balancing over packet sequence P_1 (matching rule 4 of the sample classifier in Fig. 7) followed by P_2 and P_3 (both matching rule 5): (a) hash-based load balancing may assign P_2 and P_3 to different workers so that both will have to keep the dependency list $5 \to 3 \to 1$ in the active rule set, (b) perfect separation sends P_1 and P_2 to the same worker, yielding minimal active rule sets.

at workers, which is defined as the set of rules for which a particular worker receives packets during a time window. The smaller the active rule set the fewer rules the classifier has to search through for each packet and the larger the contribution of self-adjustment to superlinear speedup. Contrarily, the larger the active rule set the more rules compete for the first positions in the list, which reduces the room for self-adjustment to reduce lookup time and erodes superlinear scaling.

There are several factors that may bloat workers' active rule sets. First, whenever a rule with nonzero dependencies is hit MRF adds its entire dependency chain to the active rule set. Second, packet classifiers often use wildcard rules, matching potentially a huge number of diverse traffic flows. If the load balancer dispatches two packets matching the same rule to two different workers, then both workers would have to include the same rule, with all its dependencies, in its active rule set (see an example in Fig. 8). Note that the same rule duplication problem plagues many software packet classifier algorithms [34, 53, 79, 87].

Designing an ideal load balancer that minimizes workers' active rule sets, regardless of rule dependencies and flow diversity, seems difficult (but see a discussion in §5). Therefore, we adopt a simple hash-based load balancing scheme here that implements only an "imperfect rule set partitioning". Our load balancer will however be fully implemented in hardware and run at line rate. This is crucial to minimize the overhead, which in our system entirely counts towards the sequential part of the workload and limits ultimate scaling. Later, we will show empirically that even this imperfect scheme is enough to reach superlinear speedup in many practical cases.

Our load balancer reuses the Receive Side Scaling (RSS, [11, 74]) function offered by most standard NICs. RSS evaluates a hash function over a selected set of header fields per each packet. The resultant hash value is then used to index into an indirection table to select a packet queue, and the corresponding CPU core, that will process the packet. The hash function can be configured to consider any combination of the IP 5-tuple header fields, which allows us to fine-tune locality-boosting in our load balancer.

4.3 Reproducing superlinear speedup

We conducted several experiments with the distributed self-adjusting packet classifier combined with the hash-based RSS load balancer. Our goal was to understand whether superlinear scaling can be robustly reproduced on a real network application using real packet I/O.

Testbed. The system-under-test (SUT) is a server equipped with a 32-core AMD EPYC 7502P@2.5 GHz CPU (64 cores with hyper-threading enabled), 128 GByte DDR4 main memory, 96 KB per-core L1 cache, 512 KB per-core L2 cache, and 128MB shared L3 cache. A server of similar configuration was used for traffic generation and measurement with DPDK/`moongen` [22], connected back-to-back to the SUT over Intel XL710 40GbE NICs. We used a standard Ubuntu 22.04.4 LTS OS running a patched v6.5 Linux kernel on the SUT, replacing the default `nftables` packet classifier with our own self-adjusting implementation. The benchmarks use the Tipsy network testing automation and visualization tool [56]. Hyper-threading was disabled, unless otherwise noted.

The classifier rule sets come from two sources. A series of *realistic rule sets* was generated with `ClassBench-ng` [59, 85], which accurately model the characteristics of real access control lists and firewalls. ClassBench uses a seed file for describing the statistics of the generated 5-tuple rules, including address ranges, port distribution, and rule dependencies. For each rule set, a matching input packet sequence was generated using the standard Classbench tools [59, 60]. We also used a series of *synthetic rule sets* and matching packet traces for conducting controlled microbenchmarks. For each rule set, we generated a matching synthetic packet trace with uniform flow-size distribution, which, recall, represents the worst-case for self-adjustment. In all cases the rules and packets using unroutable IP addresses were manually removed (otherwise, Linux would drop some packets, distorting the results). Unless otherwise noted, the benchmarks run with an RSS-based hardware load balancer using an IP 5-tuple hash.

Macrobenchmarks. First, we asked whether superlinear scaling can be reproduced with real workloads. Fig. 9a, Fig. 9b and Fig. 9c give the speedup and the raw packet rate obtained with the default `nftables` packet classifier and our self-adjusting implementation on 3 ClassBench rule sets, each containing 5000 rules, generated with the seeds `acl1`, `ipc1` and `fw1`, respectively. All rule and trace generation parameters were set to their default values.

Our observations are as follows. First, *superlinear scaling is indeed reproducible with our distributed self-adjusting packet classifier*, with maximum speedup on 32 cores ranging from 225× (about 7× faster than linear) for `acl1`, to 72× (about 2.2× of linear) with `ipc1` and 52× for `fw1` (1.6× faster than linear). In contrast, *the static `nftables` classifier scales almost linearly*. A closer analysis reveals a slow sublinear trend representative of an Amdahl's law profile for a very small sequential parameter ($s \sim 0.001$).

The speedup factor alone, however, does not reveal the full picture, as evidenced by Fig. 9c. The absolute packet rate of

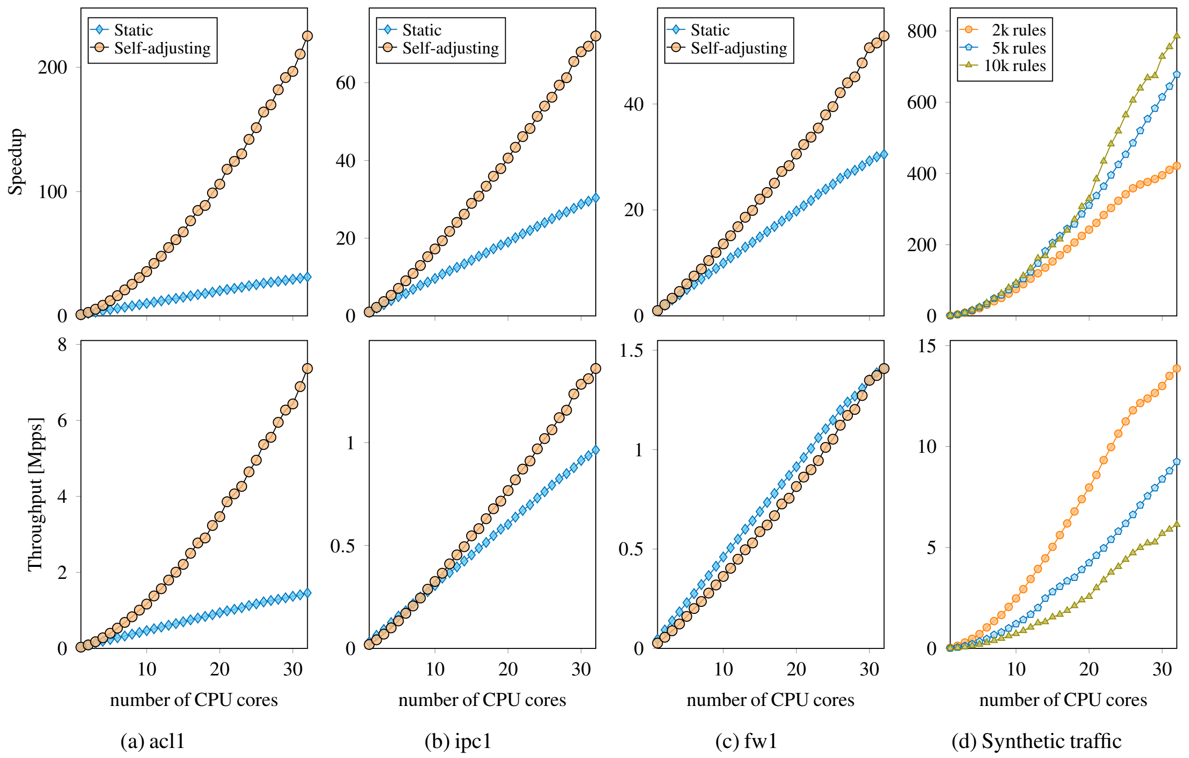


Figure 9: Scaling on ClassBench 3 rulesets generated from different seeds, containing 5000 rules each (panel (a), (b) and (c)), and synthetic rule set with uniform traffic and different rule sizes (panel (d)). Upper row shows relative speedup and the bottom row shows absolute throughput (packet rate in million packets per sec, mpps). Note the different scales on the *y* axes.

the self-adjusting classifier on the `fw1` seed is smaller than that of the static classifier, despite the superlinear speedup. In other words, a massive spurious speedup can be obtained by improving a slow baseline. Note, however, that this occurs only for the `fw1` seed (later we reveal, why); for the rest of the benchmarks the self-adjusting version is robustly faster even in terms of raw performance (5.2× for `acl1` and 1.4× with `ipc1` on 32 cores). Nonetheless, with hyperthreading enabled we obtain ∼1.5× absolute packet rate improvement on 64 cores even for the `fw1` seed (not shown in the figure), indicating that, with the sufficient amount of parallel resources, distributed self-adjustment eventually surpasses static algorithms even in terms of raw performance. In other words, when scaling is superlinear even a very slow baseline becomes extremely fast ultimately.

The mean per-packet latency is shown in Fig. 10a. We observe that *superlinear speedup transforms into massive latency reduction*, resulting 52× smaller mean packet delay on 32 cores for the `acl1` seed using the self-adjusting algorithm. In contrast, the static `nftables` classifier produces a mostly flat latency profile, stabilizing at about 13ms per packet delay.

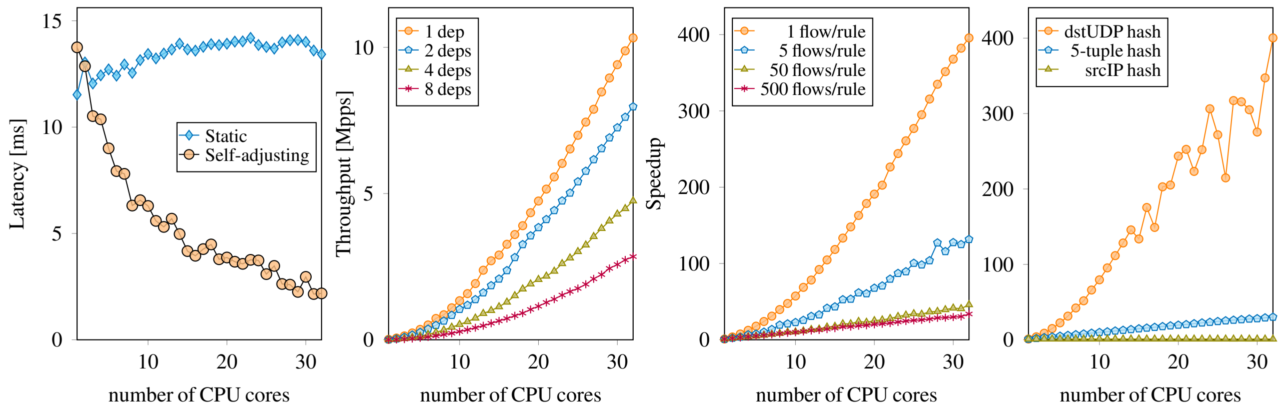
Rule size. Fig. 9d shows the speedup obtained using the self-adjusting classifier on increasingly larger rule sets. The rules were generated from the following template:

Prio	Proto	Src IP	Dst IP	Dst Port	Action
1	UDP	A.B.C.D	E.F.G.H	1	ACCEPT
2	UDP	A.B.C.D	E.F.G.H	2	DROP
...	...	...	...	...	...

The source IP and the destination IP address are the same in each rule, and each action was set to accept. We obtained 3 rule sets this way, containing roughly 2k, 5k, and 10k rules, respectively (the real size is a close prime to minimize periodicity in the scaling profiles). Note that rules are independent and each rule matches exactly one flow, which represents the optimistic case for the self-adjusting classifier (see later for the pessimistic settings). We generates a matching packet trace containing one flow per rule.

The main takeaway is that *superlinear speedup appears independently of the classifier size*, to the point that for 10k rules we see $> 800\times$ speedup on 32 cores. Again, the raw performance plot casts a complete picture: the larger the rule set the greater the superlinear speedup but the smaller the absolute packet rate. Nevertheless, superlinear scaling robustly appears in terms of the raw performance as well.

Rule dependencies. Next we turn to controlled microbenchmarks over synthetic input, which we fine-tuned to highlight the effect of some specific characteristic of the classifier workload on scaling. The main factor affecting speedup is workers' active rule set sizes, which determines the extent to which self-adjustment can arrange recently hit rules to the front of the rule list (see §4.1). Rule-dependencies have a crucial role here, since for every rule with nonzero dependencies not just the rule but all its dependencies will also become active, bloating the active rule sets.



(a) Latency (`acl1`) (b) Rule dependency (c) Active flow size (d) Locality boosting (RSS)

Figure 10: Microbenchmarks: (a) mean packet delay on the rule set generated from the `acl1` Classbench speed (5k rules, uniform traffic); (b) raw packet rate for 4 synthetic rule sets with increasingly long dependency chains; (c) speedup for 4 packet traces with increasingly more active flows per (independent rules); and (d) speedup with different RSS hash functions (same rules).

We created 3 synthetic rule sets with increasingly long dependency chains as follows. For every rule in the synthetic rule set we add an extra d overlapping rules by varying the subnet prefix length in the source IP address filter between `/32` (most specific, highest priority) and `/0` (least specific, lowest priority). This creates for every rule a chain of d increasingly more specific dependencies.

Prio	Proto	Src IP	Dst IP	Dst Port	Action
1	UDP	A.B.C.D/32	E.F.G.H	1	ACCEPT
2	UDP	A.B.C.D/31	E.F.G.H	1	DROP
...	...	...	...	...	...
...	UDP	A.B.C.D/0	E.F.G.H	1	ACCEPT
...	UDP	A.B.C.D/32	E.F.G.H	2	ACCEPT
...	...	...	...	...	...

Unfortunately, rule set size also increases d times, but this should not affect the basic superlinear speedup trends (recall Fig. 9d). We run the benchmarks with a 5k base rule set and add d dependencies per rule for $d = 1$ (small-dependency), $d = 2$, $d = 4$ and $d = 8$ (high-dependency). The packet trace contains a single flow per each "least specific" rule at the tail of the dependency chains.

Fig. 10b shows the absolute packet rate for the 4 synthetic rule sets. The most important observation is that, as expected, *the more dependencies the smaller the performance and the less visible the superlinear growth* (but note the simultaneous increase in the rule size). Manually checking the classifier statistics confirms that the MRF algorithm at each worker moves the active rules with all d dependencies to the front of the list, reducing the self-adjustment contribution to scaling for large settings of d. In terms of speedup, however, the trend is just the opposite (not shown here): the more dependencies the greater the speedup, again thanks to the slow baseline; e.g., we see $>1{,}000\times$ speedup for $d = 8$.

We also found rule-dependencies to be behind the slow scaling on the `fw1` ClassBench seed. We observed a similar slowdown when sending a huge fraction of traffic to the final "catch-all" rule specifying the default action. As this rule depends on all other rules it cannot be moved forward, degrading the self-adjusting classifier into a static list.

Flow diversity. In this microbenchmark we vary the number of flows in the input packet trace per each rule. We used the same synthetic rule set as previously, but we removed the dependencies ($n = 2$k). We generated 4 traces containing $1, 5, 50,$ and 500 uniformly distributed flows per rule, respectively. The results in Fig. 10c confirm that increasing flow diversity has negative impact on scaling: *the more flows per rule the less visible the superlinear speedup*. With a modest flow diversity (1–50 per rule) we observe 400–46$\times$ speedup on 32 cores. However, for 500 flows the superlinear trend disappears and scaling degrades to linear (32$\times$ speedup on 32 cores). We traced back the reason to the 5-tuple RSS load balancer. Recall, an optimal load balancing policy would dispatch all flows matching the same rule to the same worker, perfectly eliminating rule duplication at workers (see §4.2). However, the RSS-based 5-tuple hash only "imperfectly" partitions the rule set: manually verifying the classifier statistics reveals that for 500 flows per rule essentially every rule appears at every worker, completely removing the speedup contribution of self-adjustment, producing the linear speedup we identify in Fig. 10c.

Locality boosting. It seems that longer rule dependencies and growing flow diversity have negative impact on superlinear scaling. In this microbenchmark we show some clue that the negative impact can be removed using a proper locality boosting load balancer. In particular, Fig. 10d shows the speedup for the previous high flow-diversity benchmark (5k independent rules, 500 uniform flows per rule) with different RSS-based hash functions. Our observations are as follows. An inadequate choice for the load balancing function removes scaling altogether: e.g., the RSS hash matching on only the source IP address dispatches all input to the same worker (recall, the source IP is the same in all rules and flows), yielding no scaling at all. A better choice would be a 5-tuple hash: this at least spreads the load but, as we checked above, causes massive rule duplication across workers, constraining scaling to linear. An optimal locality-boosting load balancer, however, would dispatch the packets matching the same rule to the same worker, removing rule duplication. For our specific rule set, such "perfectly partitioning" policy is a hash function that uses only the UDP destination port. For this RSS hash, superlinear scaling is recovered in Fig. 10d, with roughly the same speedup as with no flow diversity in Fig. 10c. This confirms that faster-than-linear growth appears only if the load balancer is indeed "locality-boosting".

5 Related work

Superlinear scaling. Amdahl's famous scaling law [3], asserting sublinear speedup and diminishing returns for parallelization, is a cornerstone result in distributed computing [13, 33, 38, 38, 50]. During the almost 60 years since its first publication, various use cases were reported that seemingly violate Amdahl's scaling, triggering several useful extensions of the basic law [18, 33, 43, 58, 66]. One phenomenon that seems to have defied these attempts is faster-than-linear scaling, observed in a broad range of real workloads [24, 36, 37, 39, 40, 72, 77, 83, 84]. For instance, [39] shows superlinear speedup for PostgreSQL and traces back the reason to a new "cache plan" for caching compiled SQL queries at each thread, [72] shows that dense matrix multiplication may exhibit faster-than-linear speedup when matrix rows/columns are optimized for CPU caches, etc. Superlinear growth is often found in Nature as well, e.g., describing the scaling of human communities to large cities. [6].

There seem to be two common strategies to obtain superlinear growth [41, 72]: either do disproportionately less work per worker as system is scaled [72], or add more resources per thread [41]. These techniques, however, are difficult to apply beyond specific use cases [37] or require adding more cache space [72]. Meanwhile, there have been growing concerns whether superlinear scaling even exists in real-life applications [23]: for instance, Gunther shows that an earlier report on faster-than-linear scaling from a Hadoop MapReduce workload is attributable to a benchmarking error and, when measured the right way, reduces to sublinear scaling. This prompts the authors to conclude that superlinearity is ultimately a performance illusion, which goes against the very laws of thermodynamics. To the best of our knowledge, ours is the first universal systematic methodology that can reproduce superlinear growth in several different parallel systems.

Locality-boosting load balancing. In line with the recent trend to leverage NICs for intelligently moving data between the network, CPU, GPU and accelerators in computing systems [78], there have been several efforts to extend the static hash-based load balancing provided by RSS: Receive Flow Steering (RFS) is a mechanism to steer flows to the CPU on which the application that processes the flow is running [74] and RSS++ is a dynamic receive side scaling mechanism aiming to keep CPU load constant [11]. These mechanisms could be leveraged to implement more efficient locality boosting in the NIC: RFS can be used to direct all flows matching the same rule to the same CPU, RSS++ could be used to evenly spread load even for staggering workers that process the "difficult" high-dependency rules, etc. Furthermore, Reframer can be used to reorder packets for improving the temporal locality at workers' input [26, 52], and SAX-PAC can be used to decompose a classifier rule set with many dependencies into multiple smaller but independent rule sets [49]. Hicuts [34], Hypercuts [79], Efficuts [87], and CutSplit [53] define "intelligent" packet header space cuts [48] to partition a rule set along a decision tree into smaller rule lists stored in the leaves of the tree. These schemes are complimentary to our approach: while [34, 53, 79, 87] use "smart" cuts with "dumb" lists in the leaves we rather use "dumb" cuts, implemented by hash-based load balancing, with "smart" rule lists in the workers to reach superlinear parallel scaling.

Self-adjusting data structures. Self-adjusting algorithms, the other ingredient for superlinear scaling, are widely applied in computer systems: caches are extensively used in predictive NFV state stores [51], database accelerators [24, 27, 63], distributed web caching and CDNs [90], and microservices [89]; move-to-front (MTF) lists are used for computing point maxima and convex hulls [14], program compilation and interpretation [42], detecting collisions in hash tables [42], and data compression [15]; further examples are splay trees [81], self-adjusting skip lists [16], push-down trees [8], or self-adjusting geometric data stores [70], etc. All these are candidates to be used, along with a proper locality-boosting load balancer, to reach superlinear scaling in distributed applications; to what extent these algorithms *already* achieve superlinear scaling in production applications is perhaps one of the most intriguing open questions for future research.

6 Conclusions

In this paper, we present theoretical and empirical proof that locality-boosting load balancing combined with parallel self-adjusting algorithms together yield faster-than-linear speedup in many applications on a wide range of workloads. Our main contribution not necessarily stands in that we show that superlinear speedup *exists* (this has been known for quite some time), but rather that we identify the main architectural patterns commonly appearing in the use cases where it was genuinely observed and synthesize these into a comprehensive and universal design methodology to *reproduce* it. This methodology is then used in extensive simulations, producing an order of magnitude faster scaling than previously observed. We extend the default `nftables` Linux subsystem into a true self-adjusting packet classifier, which we use to identify the main workload characteristics (rule-dependency, flow diversity) that affect superlinear growth trends. Future research will be needed to apply our methodology in a broader range of use cases: for instance, rule-based network intrusion detection systems like Snort or Suricata [44] or explainable AI inferencing seem like appealing application candidates.

References

[1] V. Addanki, M. Pacut, A. Pourdamghani, G. Rétvári, S. Schmid, and J. Vanerio. Self-adjusting partially ordered lists. In *IEEE Conference on Computer Communications*, IEEE INFOCOM, pages 1–10, 2023.

[2] S. Albers and S. Lauer. On list update with locality of reference. *J. Comput. Syst. Sci.*, 82(5):627–653, 2016.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485. Association for Computing Machinery, 1967.

[4] G. Andersson. An approximation algorithm for max p-section. In *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science*, pages 237–247, 1999.

[5] K. Andreev and H. Räcke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006.

[6] S. Arbesman, J. M. Kleinberg, and S. H. Strogatz. Superlinear scaling for innovation in cities. *Phys. Rev. E*, 79, 2009.

[7] C. Avin, M. Bienkowski, A. Loukas, M. Pacut, and S. Schmid. Dynamic balanced graph partitioning. *SIAM J. Discret. Math.*, 34(3):1791–1812, 2020.

[8] C. Avin, K. Mondal, and S. Schmid. Dynamically optimal self-adjusting single-source tree networks. In *LATIN 2020: Theoretical Informatics - Latin American Symposium*, volume 12118 of *Lecture Notes in Computer Science*, pages 143–154, 2020.

[9] S. Bai, H. Zheng, C. Tian, X. Wang, C. Liu, X. Jin, F. Xiao, Q. Xiang, W. Dou, and G. Chen. Unison: a parallel-efficient and user-transparent network simulation kernel. In *European Conference on Computer Systems*, EuroSys, page 115–131, 2024.

[10] A. Bar-Noy and M. Lampis. Online maximum directed cut. *J. Comb. Optim.*, 24(1):52–64, 2012.

[11] T. Barbette, G. P. Katsikas, G. Q. Maguire, and D. Kostić. RSS++: load and state-aware receive side scaling. In *International Conference on Emerging Networking Experiments And Technologies*, ACM CoNEXT '19, page 318–333, 2019.

[12] T. Barbette, E. Wu, D. Kostić, G. Q. Maguire, P. Papadimitratos, and M. Chiesa. Cheetah: a high-speed programmable load-balancer framework with guaranteed per-connection-consistency. *IEEE/ACM Transactions on Networking*, 30(1):354–367, 2022.

[13] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *Computer*, 39(1):110–112, 2006.

[14] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica*, 9(2):168–183, 1993.

[15] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986.

[16] P. Bose, K. Douïeb, and S. Langerman. Dynamic optimality for skip lists and b-trees. In *ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 1106–1114, 2008.

[17] B. Burns. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly Media, Inc., 1st edition, 2018.

[18] K. W. Cameron and R. Ge. Generalizing Amdahl's Law for power and energy. *Computer*, 45(3):75–77, 2012.

[19] F. Chang, W. chang Feng, and K. Li. Approximate caches for packet classification. In *IEEE INFOCOM*, volume 4, pages 2196–2207, 2004.

[20] F. K. Došilović, M. Brčić, and N. Hlupić. Explainable artificial intelligence: A survey. In *International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 210–215. IEEE, 2018.

[21] 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 *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Mar. 2016.

[22] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[23] V. Faber, O. M. Lubeck, and A. B. White. Superlinear speedup of an efficient sequential algorithm is not possible. *Parallel Comput.*, 3(3):259–260, 1986.

[24] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, 2004.

[25] A. Frieze and M. Jerrum. Improved approximation algorithms for MAX k-CUT and MAX BISECTION. In *Algorithmica 18*, pages 67–81, 1997.

[26] H. Ghasemirahni, T. Barbette, G. P. Katsikas, A. Farshin, A. Roozbeh, M. Girondi, M. Chiesa, G. Q. M. Jr., and D. Kostić. Packet order matters! improving application performance by deliberately delaying packets. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 807–827, 2022.

[27] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller. BMC: accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501, 2021.

[28] hashicorp/btree. https://pkg.go.dev/github.com/google/btree.

[29] golang-collections/collections. https://pkg.go.dev/github.com/golang-collections/collections#readme-splay-tree.

[30] hashicorp/golang-lru. https://pkg.go.dev/github.com/hashicorp/golang-lru/v2.

[31] N. Gunther. Superlinear scalability. In *Symposium and Bootcamp on the Science of Security*, HotSoS '13, 2013.

[32] N. Gunther, P. Puglia, and K. Tomasette. Hadoop superlinear scalability: The perpetual motion of parallel performance. *Queue*, 13(5):20–42, 2015.

[33] N. J. Gunther. *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[34] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[35] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[36] M. Gusev and S. Ristov. Superlinear speedup in Windows Azure cloud. In *IEEE International Conference on Cloud Networking (CLOUDNET)*, pages 173–175, 2012.

[37] J. Gustafson. Fixed time, tiered memory, and superlinear speedup. In *Distributed Memory Computing Conference*, volume 2, pages 1255–1260, 1990.

[38] J. L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, may 1988.

[39] R. Haas. Scalability, in graphical form, analyzed. `http://rhaas.blogspot.com/2011/09/scalability-in-graphical-form-analyzed.html`, 2011.

[40] H. Hamann. Superlinear scalability in parallel computing and multi-robot systems: Shared resources, collaboration, and network topology. In *Architecture of Computing Systems (ARCS 2018)*, pages 31–42. Springer, 2018.

[41] D. Helmbold and C. McDowell. Modelling speedup (n) greater than n. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):250–256, 1990.

[42] J. H. Hester and D. S. Hirschberg. Self-organizing linear search. *ACM Comput. Surv.*, 17(3):295–311, 1985.

[43] M. D. Hill and M. R. Marty. Amdahl's Law in the multicore era. *Computer*, 41(7):33–38, 2008.

[44] H. Jiang, G. Zhang, G. Xie, K. Salamatian, and L. Mathy. Scalable high-performance parallel design for network intrusion detection systems on many-core processors. In *Symposium on Architectures for Networking and Communications Systems*, ACM/IEEE ANCS, page 137–146, 2013.

[45] D. W. Jones. Application of splay trees to data compression. *Communications of the ACM*, 31(8):996–1007, 1988.

[46] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 97–112, 2017.

[47] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, Apr. 2018.

[48] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Symposium on Networked Systems Design and Implementation*, USENIX NSDI, pages 113–126, 2012.

[49] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. SAX-PAC (Scalable And EXpressive PAcket Classification). In *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, page 15–26, 2014.

[50] S. Krishnaprasad. Uses and abuses of Amdahl's Law. *J. Comput. Sci. Coll.*, 17(2):288–293, dec 2001.

[51] J. Lei and V. Shrivastav. Seer: Enabling Future-Aware online caching in networked systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 635–649, 2024.

[52] T. Lévai, F. Németh, B. Raghavan, and G. Retvari. Batchy: batch-scheduling data flow graphs with service-level objectives. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 633–649, 2020.

[53] W. Li, X. Li, H. Li, and G. Xie. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM*, pages 2645–2653, 2018.

[54] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast In-Memory Key-Value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Apr. 2014.

[55] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. Survey of performance acceleration techniques for Network Function Virtualization. *Proceedings of the IEEE*, 107(4):746–764, 2019.

[56] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári. The price for programmability in the software data plane: The vendor perspective. *IEEE Journal on Selected Areas in Communications*, 36(12):2621–2630, 2018.

[57] S. Mahajan and J. Ramesh. Derandomizing semidefinite programming based approximation algorithms. In *Proc. 36th Ann. IEEE Symp. on Foundations of Comput. Sci. (FOCS)*, pages 162–169, 1995.

[58] A. Marowka. Extending Amdahl's Law for heterogeneous computing. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 309–316, 2012.

[59] J. Matoušek, G. Antichi, A. Lučanský, A. W. Moore, and J. Kořenek. ClassBench-ng: recasting ClassBench after a decade of network evolution. In *Symposium on Architectures for Networking and Communications Systems*, IEEE/ACM ANCS, page 204–216, 2017.

[60] S. Miano. sebymiano/pcap-utils. `https://github.com/sebymiano/pcap-utils`.

[61] S. Miano, A. Sanaee, F. Risso, G. Rétvári, and G. Antichi. Domain specific run time optimization for software data planes. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1148–1164, 2022.

[62] The nftables project. `https://wiki.nftables.org`.

[63] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Apr. 2013.

[64] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, 2018.

[65] ONF. Openflow reference release. `https://github.com/mininet/openflow`, 2013.

[66] I. Onyuksel and S. Hosseini. Amdahl's law: a generalization under processor failures. *IEEE Transactions on Reliability*, 44(3):455–462, 1995.

[67] M. H. Overmars and F. A. van der Stappen. Range searching and point location among fat objects. *J. Algorithms*, 21(3):629–656, 1996.

[68] M. Pacut, J. Vanerio, V. Addanki, A. Pourdamghani, G. Rétvári, and S. Schmid. Self-adjusting packet classification. *CoRR*, abs/2109.15090, 2021.

[69] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for NFV applications. In *Symposium on Operating Systems Principles*, SOSP '15, page 121–136, 2015.

[70] E. Park and D. M. Mount. A self-adjusting data structure for multidimensional point sets. In *Algorithms - ESA Annual European Symposium*, volume 7501, pages 778–789, 2012.

[71] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[72] S. Ristov, R. Prodan, M. Gusev, and K. Skala. Superlinear speedup in HPC systems: Why and when? In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898, 2016.

[73] O. Rottenstreich and J. Tapolcai. Optimal rule caching and lossy compression for longest prefix matching. *IEEE/ACM Transactions on Networking*, 25(2):864–878, 2016.

[74] Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[75] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808, 2021.

[76] C. Sears. The elements of cache programming style. In *4th Annual Linux Showcase & Conference (ALS 2000)*. USENIX Association, 2000.

[77] Sdn analytics and control: Superlinear. https://blog.sflow.com/2010/09/superlinear.html, 2010.

[78] J. Sherry. The I/O driven server: From SmartNICs to data movement controllers. *Computer Communications Review (CCR)*, 53(3), 2023.

[79] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, page 213–224, 2003.

[80] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, Feb. 1985.

[81] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[82] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu. Nfp: Enabling network function parallelism in nfv. In *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 43–56, 2017.

[83] H. Sutter. Going superlinear. Dr. Dobb's J., 2008. https://www.drdobbs.com/cpp/going-superlinear/206100542.

[84] H. Sutter. Super linearity and the bigger machine. Dr. Dobb's J., 2008. https://www.drdobbs.com/parallel/super-linearity-and-the-bigger-machine/206903306.

[85] D. E. Taylor and J. S. Turner. ClassBench: a packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3):499–511, 2007.

[86] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff. Revisiting the Open VSwitch Dataplane ten years later. In *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '21, page 245–257, 2021.

[87] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: optimizing packet classification for memory and throughput. In *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '10, page 207–218, 2010.

[88] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker. Elastic scaling of stateful network functions. In *USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 299–312, 2018.

[89] H. Zhang, K. Kallas, S. Pavlatos, R. Alur, S. Angel, and V. Liu. MuCache: A general framework for caching in microservice graphs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 221–238, 2024.

[90] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. Rashmi. SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1229–1246, 2024.

A Analytical findings

Scaling in distributed systems refers to the improvement in performance achieved by adding more machines to a system, defined as the ratio of the completion time of the baseline system with a single machine to the completion time of the system with k machines, $S(k) = T(1)/T(k)$.

We claim that our system scales along two dimensions: load balancer efficiency $\ell(k)$ and parallelizability $q(k)$.

$$S(k) = \frac{T(1)}{T(k)} = \frac{1}{s + \frac{1-s}{q(k)\cdot\ell(k)}} \ ,$$

for some values of $q(k)$ and $\ell(k)$ that depend on the input σ and the load balancer. The value $\ell(k)$ captures the reduction in the total work of the system by using a load balancer, and $q(k)$ captures how well the reduced workload can be parallelized on k machines. If $q(k)\cdot\ell(k) > k$, we say that our system *scales superlinearly*.

We draw the following conclusions from our analysis.

1. The system can scale superlinearly for certain input streams combined with the right load balancer. The uniform input example is just one such example.

2. On the contrary, some other input streams cannot even achieve linear scaling with any load balancer.

3. The paralellization factor $q(k)$ can be at most k (reaching $q(k) = k$ for uniform input).

4. The workload reduction factor $\ell(k)$ depends on the input sequence σ and the load balancer.

5. The workload reduction factor $\ell(k)$ cannot be reduced indefinitely with growth of k. Hence, the system can scale superlinearly only for small values of k.

Our analysis holds for a wide range of self-adjusting data structures (including LRU caches).

B The model

Architecture. Consider k identical parallel machines $M_1, M_2, M_3, \ldots, M_k$, each having its own isolated memory and running an instance of a self-adjusting list D. The stream of requests σ arriving at a load balancer is partitioned into k streams $\sigma(M_1), \sigma(M_2), \ldots, \sigma(M_k)$ and dispatched to the machines. The load balancer dispatches the requests to machines based solely on the request itself, and not on the state of the system. The load balancer is a function $f_k : \mathcal{U} \to \{1,2,\ldots,k\}$ from the universe of all items $\mathcal{U}$ to the machines, and this function partitions the universe $\mathcal{U}$ into k subsets $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_k$ (often referred to as affinity domains).

Cost model. The time to process a request σ_t at time t includes both the computational overhead of the load balancer (denoted $T(f_k(\sigma_t))$) and the processing time by D at machine $M_{f_k(\sigma_t)}$ (denoted $T(g_D(\sigma_t))$). Notably, the processing time g_D varies over time due to the self-adjusting nature of the data structure.

Self-adjusting data structures often achieve some variant of *working set property* that links the input history to the cost of processing a request. In our work, the working set for an item σ_t request at time t is defined as the set of distinct items requested since the last request to the item σ_t. With each data structure D, there exists an associated cost function g_D

$$\mathsf{cost}(x,t,\sigma) \le g_D(|W_t(x)|) \ ,$$

where $W_t(x)$ is the number of distinct requests to items other than x since the last request to x. With these assumptions, we capture e.g. LRU caches, Move-to-Front lists, splay trees and more.

Objective. Our goal is to minimize the completion time of the schedule of jobs induced by the stream of requests σ executed on parallel machines. The schedule finishes when all requests from σ are processed. The load may be uneven, and some machines may be idle throughout execution, but the system is not allowed to reassign the requests to other machines.

Benchmark. Our benchmark $T(1)$ is a single self-adjusting data structure D that runs on a single machine M_1 and processes the entire stream σ, with a trivial load balancer $f_1(\cdot) = M_1$. A single self-adjusting data structure is the most natural baseline choice for self-adjusting data structures.

C Superlinear Scaling of Self-adjusting Distributed Systems (a positive result)

The load balancer f_k partitions the input stream σ into k more local streams $\sigma(M_1), \sigma(M_2), \ldots, \sigma(M_k)$, and reduces the sum of machine's workloads by a factor of $\ell(k)$. The workload is then executed on k machines, with the objective to reduce the schedule completion time, which brings speedup of the factor of $q(k)$, $q(k) \le k$.

C.1 Study of $\ell(k)$: how workload is reduced

The value of $\ell(k)$ depends on the input σ and the load balancer f_k. Hence, $\ell(k)$ indirectly relies on k through f_k (these are tied together in our architecture). Furthermore, for technical reasons, $\ell(k)$ depends on the serial portion of the workload s. To estimate $\ell(k)$ for a given stream σ, we need to relate σ with the load balancer f_k for the given data structure D.

Self-adjusting data structures and working sets. Our law captures various self-adjusting data structures, such as lists, caches and their generalizations. In these data structures, the cost of accessing an item depends on the internal structure and changes over time depending on the history of requests. Self-adjusting data structures have a property that the cost of accessing an item x at time t depends on the number of distinct items requested since the last access of x. This is often referred to as the *working set property*, and it can hold in an amortized sense. A working set for an item σ_t request at time t is defined as the set of distinct items requested since the last request to the item σ_t.

With each data structure D, there exists an associated cost function g_D

$$\mathsf{cost}(x,t,\sigma) \leq g_D(|W_t(x)|) \quad ,$$

where $W_t(x)$ is the number of distinct requests to items other than x since the last request to x. With these assumptions, we still capture parallel extensions of LRU caches, Move-to-Front lists, splay trees.

We illustrate g_D for Move-to-Front and LRU. In Move-to-Front the cost of accessing an item is linear: $g_{\mathsf{MTF}}(x,t,\sigma) = |W_t(x)| + 1$. LRU is the algorithm Move-to-Front casted into the cost model of caching with a generalized cost function g_{LRU} that is non-linear: for a cache of size B, the cost is 1 if the working set size is $B+1$, and 0 otherwise. We note that this generalized setting introduced by Sleator and Tarjan [80] captures more general data structures than just lists and caching under a common characterization of g_D.

Load balancer isolates working sets. Recall that the load balancer partitions the stream into k streams $\sigma(M_1),\sigma(M_2),\ldots,\sigma(M_k)$ and dispatches them to the machines. These streams may have reduced working set sizes at the machines in comparison to the working set sizes of the original stream σ. Precisely, a load balancer $f_k : \mathcal{U} \rightarrow \{1,2,\ldots,k\}$ partitions the universe $\mathcal{U}$ into k subsets $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_k$, and the working set for machine i at the time $x = \sigma_t$ is requested is $W_t(x,t,\sigma(M_i)) = W_t(x,t,\sigma) \cap \mathcal{U}_i$.

The cost for the parallel workload of the baseline is

$$T(1) = \sum_t g(|W_t(x)|) \quad .$$

The cost of the parallel workload for the distributed system is as follows. In total, we observe cost savings from load balancing for the stream σ, data structure D characterized by a function g, and a load balancer f_k.

$$T(k) = \sum_t g(|W_t(x) \cap \mathcal{U}_{f_k(\sigma_t)}|) \quad .$$

Note the total work decreases[1] by r for the stream σ.

The speedup ratio in the dimension of $\ell(k)$ is then

$$1/\ell(k) = \frac{\sum_t g(|W_t(x) \cap \mathcal{U}_{f_k(\sigma_t)})|}{\sum_t g(|W_t(x)|)} \quad .$$

[1]A strict deviation from the assumptions of Amdahl's law, where workload is fixed. See Section C.4 for a discussion on our assumptions.

C.2 The study of $q(k)$: how parallelizable the reduced workload is

Recall that our objective is not to minimize the total work, but to minimize the completion time of the schedule. Fix a reduced parallel workload from the previous section of total size $(1-s)/\ell(k)$, and let's look at its components. We are interested in minimizing the completion time of the last machine. The speedup ratio in the dimension of $q(k)$ is given by

$$1/q(k) = \frac{\sum_t g(|W_t(x)) \cap \mathcal{U}_{f_k(\sigma_t)|}}{\max_i \sum_{t: f_k(\sigma_t)=i} g(|W_t(x) \cap \mathcal{U}_{f_k(\sigma_t)}|)} \quad .$$

Some request streams and load balancer pairs are better at achieving $q(k)$ close to its theoretical limit k (given by Amdahl's law). Our uniform input example achieves perfect parellelization of $q = k$, since all jobs are the same size and all machines process the same number of jobs. On the other hand, some unparallelizable streams such as a repeated request to a single item have $q = 1$, because they use only one machine in our architecture[2]. Heterogeneous workload can also be parallelizable, for example a single machine M_1 can process a majority of the stream σ if the stream σ_{M_1} is local, while the rest of the machines process longer jobs to finish at the same time as M_1. It should however be obvious that the streams that achieve good parallelization $q(k)$ need to have roughly equal workloads for each machine, and the only streams that can achieve that consist of requests to multiple affinity domains. To maximize $q(k)$, the load balancer needs to distribute the workload evenly among the machines.

C.3 Characterizing the speedup

Definition 1. *Fix any stream σ and load balancer f_k. The value $\ell(k)$ is defined as the ratio of the total parallel work of the distributed system to the total parallel work of the baseline.*

Definition 2. *Fix any stream σ and load balancer f_k. The value $q(k)$ is defined the ratio of the total reduced parallel work $(1-s) \cdot \ell(k)$ to the completion time of the last machine.*

Then, the speedup of the distributed system is given by the following theorem.

Theorem 1. *Consider a data structure D with a cost that is upper-bounded by a non-decreasing function g_D of the working set size.*

Consider a load balancer that dispatches inputs σ taken from a universe $\mathcal{U}$ to k identical parallel workers $W_1, W_2, \ldots, W_k$, each running an instance of a self-adjusting algorithm D, using a deterministic function $f_k : \mathcal{U} \rightarrow \{1,2,\ldots,k\}$ that partitions the input universe $\mathcal{U}$ into k disjoint subsets $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_k$. For an input σ, self-adjusting distributed systems scaling with k

[2]To remedy that, we could look into load balancers that distribute jobs to multiple machines (generalizations of functions f_k, to functions from $\mathcal{U}$ to sets of machines). This goes beyond the scope of this paper. See RSS+ paper for reference [11].

is characterized in two dimensions: load balancer efficiency $\ell(k)$ and parallelizability $q(k)$. This gives us speedup

$$\frac{T(1)}{T(k)} \leq \frac{1}{s+\frac{1-s}{q(k)\cdot\ell(k)}} \ ,$$

for $q(k)$ and $\ell(k)$ defined in Definition 2 and 1, which depend on the input σ, load balancer f_k and the cost function g_D.

The proof of the above theorem is a direct consequence of executing a reduced workload (reduced by a factor of $\ell(k)$) on k machines with the parallelization $q(k)$. Note that s is the serial fraction of the workload, which is common to $T(1)$ and $T(k)$.

Our theorem applies to e.g. Move-to-Front lists and LRU caches, since their cost functions g_{MTF} and g_{LRU} are monotonically increasing in the working set size.

We dedicate the next two subsections to partitioning the speedup into these two dimensions and independently analyzing them.

C.4 Discussion

Deviating from Amdahl's law assumptions. Traditionally, scaling has been upper-bounded using Amdahl's law, which predicts that the overall performance improvement is limited by the fraction of the workload that cannot be parallelized. Amdahl's law holds under the assumption that the problem size is fixed, the workload is divisible, the tasks are independent, and we use a baseline that computes the fixed size problem on a single machine. With these assumptions, no system can scale better than linearly. We deviate from these assumptions, (1) the problem size at the machines can decrease, and (2) the workload is not perfectly divisible, and (3) our baseline is a single self-adjusting data structure running on a single machine that processes the entire stream σ. In particular, our problem size is not fixed, but rather the total work decreases for k machines in comparison to the work accomplished by the baseline.

One may ask why our perspective is useful. In our eyes, for systems consisting of self-adjusting data structures, the most natural benchmark is a single self-adjusting data structure. This perspective explains the superlinear scaling observed in practice for caching.

D Impossibility of scaling for self-adjusting distributed systems (a negative result)

We finalize by sketching a scaling impossibility law, an equivalent of Amdahl's law for self-adjusting distributed systems. We draw conclusions that for each stream σ the scaling can be only initial (cannot continue indefinitely with k), hence the superlinear scaling observed in practice is only an ephemeral phenomenon.

Theorem 2. *Assume that the cost of a data structure D is lower-bounded in terms of the working set size as a monotonically non-decreasing function g_D. Then, our distributed self-adjusting system cannot scale better than*

$$S(k) = \frac{T(1)}{T(k)} = \frac{1}{s+\frac{1-s}{k\cdot\ell(k)}} \ ,$$

where $\ell(k)$ depends on g_D, σ and f_k.

The proof of this theorem is a direct consequence of the Amdahl's law to the reduced workload: the reduced parallel workload $(1-s)/\ell(k)$ can be executed at most k times faster. We leave the proof of to the full version of the paper.

There are two consequences of the above theorem.

1. **Superlinearity is an initial-only phenomenon**

   The maximum possible multiplicative $\ell(k)$ achievable is fixed for each input sequence σ. This always happens when k is large enough so the load balancer f_k isolates all working sets, but for many inputs σ, the improvements can dry up even for smaller k (the more local the sequences are, the more effective the load balancer can be). As a corollary, the system can still scale with the parameter k due to additional resources, but the savings from load balancing do not increase indefinitely with k.

   Observation 1. *For each input sequence σ, there exists a constant K such that for all $k > K$, for each load balancer f_k, the savings $\ell(k)$ are fixed do not increase with k. Combined with the fact that $q \leq k$, means that superlinear scaling cannot continue with k indefinitely.*

2. **Tight analysis for LRU caches and MTF lists**

   For Move-to-Front lists and LRU caches, the cost function $\ell(k)$ is both upper and lower bounded as a function of the working set size. Hence, our analysis of scaling is tight for these algorithms.

   In particular, the LRU algorithm for caching cannot scale superlinearly indefinitely. Hence, the observed phenomenon in the literature is only an initial phenomenon due to the combined effect of increased number of machines and the load balancer.

E How to find good load balancers?

There are many constraints for the load balancer, e.g. it should be efficiently computable and should parallelize the workload well (measured by the parameter q). Now, we focus solely on the locality-boosting aspect of the load balancer, how well the workload is reduced by cutting affinity domains with f_k. We can visualize the saving from load balancing with help of a weighted complete graph, where each edge weight represents the saved cost by isolating the affinity domains of the two machines. For each input stream, the costs of a self-adjusting data structure can be decomposed into the sum of costs accounted to pairs of nodes, see the work of Albers and Lauer for details [2]. The optimal load balancer uses the heaviest cut in such a graph.

First, we consider load balancers that remain fixed over time, and we additionally assume that the input stream is known in advance. The optimal k-cut is known as *maximum k-cut*, and is known to be NP-hard problem [25, 57]. If the cuts are balanced (a natural choice), then the problem is known as *maximum k-section* [4] (from the perspective of maximizing the cut) and *minimum graph k-balanced partitioning* [5] (from the perspective of minimizing the non-cut edges). In the former model, we have a polynomial time algorithm that achieves a constant-factor approximation [4], and in the latter model, we have a polylogarithmic approximation [5].

It is often unrealistic to know the entire input sequence in advance, and online variants of the problem are studied, where additionally the load balancer can change the assignment of the requests to the machines over time. We refer to online variants of the above problems: online k-cut [10] and online graph partitioning [7].