More Bang for the Buck: Superlinear Scaling with Distributed Self-Adjusting Systems

Jonas Köppeler
TU Berlin, Germany

Maciej Pacut
TU Berlin, Germany

Tamás Lévai
Budapest University of Technology and Economics, Hungary

Vamsi Addanki
Purdue University, West Lafayette, IN, USA

Stefan Schmid
TU Berlin, Germany

Gábor Rétvári
Budapest University of Technology and Economics, Hungary

Abstract

Extracting maximum performance from a limited pool of parallel compute resources remains a central challenge. In this paper, we show an optimization technique that allows certain distributed systems to attain faster-than-linear (superlinear) performance improvement with only a linear scaling of the worker pool. Our insight is that (1) dispatching jobs to parallel workers so that the locality of reference in the workers' input increases and (2) implementing the workers with a self-adjusting algorithm to take advantage of the higher locality can yield superlinear scaling in many practical applications. First, we demonstrate our technique in simulations: by scaling textbook self-adjusting algorithms, we obtain 100–3,300x speedup using only 48 CPU cores – up to 70x beyond linear scaling. After that, we re-engineer the default Linux packet classifier to attain a 5–25x raw performance improvement as compared to the vanilla kernel. We demonstrate 800x speedup on synthetic traces and 220x speedup on real firewall traces with 32 CPU cores. Given these insights, we develop a formal model and a set of design guidelines to help understand the applicability of our optimization strategy for particular distributed system workloads.

2012 ACM Subject Classification Networks → Network algorithms; Theory of computation → Design and analysis of algorithms

Keywords and phrases self-adjusting systems, superlinear scaling, packet classification

Digital Object Identifier 10.4230/LIPIcs.SAND.2026.3

Category Invited Talk

Supplementary Material *Software (Source Code)*: `https://github.com/inet-tub/Superlinear-Scaling-with-Distributed-Self-adjusting-Systems`
archived at `swh:1:dir:53f5b299bb898841ac4c3c775cb1199f42034de2`

Funding European Research Council (ERC), Proof-of-Concept grant 101287293 (FortifyNet), 2026–2027.

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 [50, 71, 78, 85, 92].

In horizontally scaled applications a load balancer (LB) dispatches jobs across a fleet of workers that process the jobs in parallel [19]. In the context of *web applications*, HTTP load balancers [13,23,68] distribute requests across a swarm of backend web servers. Multicore *OS network stacks* [12,51,89] run multiple instances of the networking logic on different CPUs and leverage the NIC to dispatch packets to CPU cores. In *sharded key-value stores* [30] different servers handle different portions of the key-space.

Suppose a web app handles 100 requests per second using a single server. As we add another server we expect the throughput to increase to 200 requests per second, or slightly less if the system is not perfectly parallel [4]. (Sub)linear scaling feels evident in this context: as the additional capacity can be consumed at 100 percent efficiency at best, we can "get at most equal bang for the capacity buck" [35]. Curiously, several experiments reported faster-than-linear (or *superlinear*) growth in certain high-performance computing applications and distributed systems [10,18,26,40,41,43–45,74,79,80,86,87]. Superlinear growth feels particularly alluring in this context, in that it assumes a system that somehow manages to produce more work than the computer capacity available to it [35].

In fact, faster-than-linear scaling is not supernatural at all. Suppose our sample web app serves a set of static assets (web pages, images, etc.) and assume each server is assigned a fixed subset of the assets, with a load balancer carefully routing client requests for each asset to the proper server. As the number of servers increases each server perceives requests to a progressively smaller subset of the assets, which may allow it to finish servicing requests faster, say, by *caching* the most popular assets in fast memory [21,26,79]. The combined speedup resulting from the increase of web server capacity and the decrease of servers' "virtual job size" due to improved cache efficiency often yields superlinear scaling [18,41,41–43,45,74,86,87,91]. This is, however, extremely sensitive to subtle design choices and a poor implementation can easily destroy the delicate superlinear scaling trend (see § 2).

Although superlinearity is thoroughly analyzed [35,41,45,74,86,87] and evaluated [10, 18,40,43,44,80] in piecemeal applications, currently no architectural blueprint exists to *methodologically optimize distributed systems towards superlinear scaling.*

In this paper, we aim to contribute towards closing this gap by identifying a design pattern and the conditions under which it leads to superlinear scaling. Our motivation is that networking applications are often embarrassingly parallel with little or no dependency between threads, promising massive (superlinear) parallel execution gains.

We observe that in order to achieve superlinearity one has to carefully combine an appropriate load balancing policy with a proper worker implementation. Indeed, load balancing in distributed systems is often non-arbitrary: web apps apply the "sticky sessions" rule to route all requests of a particular user to the same web server; networking code commonly uses IP 5-tuple hashing at the NIC to ensure that all packets of a flow are processed on the same CPU; and key-hashing in sharded key-value stores concentrates queries to a key at the same replica. Such policies tend to make the input streams processed by the parallel workers more predictable, compared to the aggregate input processed by the system. Combining such a *locality boosting load balancer* with a *self-adjusting algorithm* so that workers can take advantage of the higher input predictability to adaptively improve their own performance yields faster-than-linear speedup across several applications.

The power of this optimization technique (§ 3) is *not* that it confirms the existence of superlinear scaling (this has been known for a while [86,87]), neither that it defies well-established scaling laws (it does not, see [25,34,35,45]) nor that it produces the most efficient implementations possible (e.g., our packet classifier will not be as efficient as, say, a DPDK equivalent [77] just by the fact that it runs inside the Linux kernel [27]). Rather, our

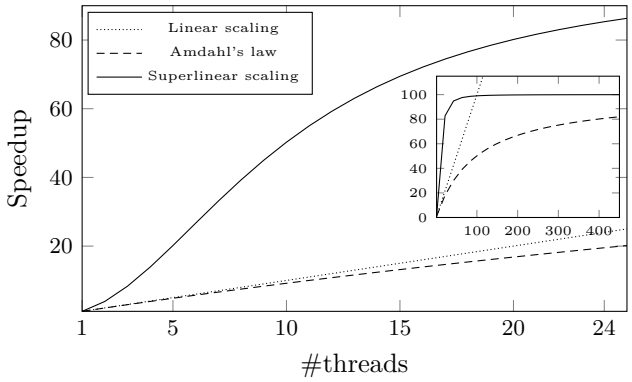


Figure 1 Linear scaling, Amdahl's law and superlinear scaling ($s = 0.01$). The inset shows the asymptotics.

main contribution is that we precisely identify the key ingredients – locality boosting and self-adjustment – that, in combination, form a reusable design pattern to attain superlinear scaling under specific conditions, enabling re-engineering of common distributed systems with little effort to attain often orders-of-magnitude performance improvement.

First, we confirm the viability of our approach in extensive simulations. Deploying well-known list and tree search algorithms from the literature we achieve 100–3300× speedup on 48 CPU cores, orders of magnitude surpassing plain, linear scaling. We support our empirical findings with a formal analysis and obtain a new scaling law for distributed self-adjusting systems (Appendix A). Then we present two fully operational case studies. As a major contribution we re-engineer the packet classifier built into the popular Linux kernel to reach superlinear scaling (§ 4). On synthetic and real-life firewall traces, our implementation exhibits up to 800× speedup with 32 CPU cores, 5–25× improvement beyond the default Linux firewall implementation which scales only (sub)linearly. We also identify the key metrics (rule dependencies and flow diversity) that critically determine the achievable performance improvement for a specific workload (§ 5). Furthermore, we apply our methodology to a combined Memcached+PostgreSQL storage system, yielding 2.3× faster than linear scaling (discussions moved to Appendix B for space reasons). We finally review related work (§ 6) and summarize our findings (§ 7). The self-adjusting Linux `nftables` implementation is available at `https://github.com/inet-tub/Superlinear-Scaling-with-Distributed-Self-adjusting-Systems`.

2 Background

We use "distributed" and "parallel" interchangeably to refer to a system running across multiple independent compute threads ("workers"), whether on parallel CPUs within a single node or across multiple nodes.

Amdahl's law [4] sets a fundamental limit on the speedup achievable by parallelizing a program. Let s be the fraction of execution time spent in code that cannot be parallelized (e.g., single-threaded sections, critical sections with locks), and $(1 - s)$ the parallelizable fraction. Denote by $T(k)$ the runtime and by $S(k) = \frac{T(1)}{T(k)}$ the speedup on k processors. Then the following holds (see Fig. 1):

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

Here, $\frac{1-s}{k}$ establishes that the perfectly parallel part of the program executes k times faster on k processors than on a single core. For different applications and extensions of Amdahl's law, see [14, 20, 36, 37, 47, 62, 70].

Amdahl's law suggests that parallel systems can scale linearly at best ($\frac{dS(k)}{dk} \leq 1$, with equality exactly when $s = 0$). Curiously, there have been several reports from a broad range of applications indicating faster-than-linear scaling, e.g., database systems [26, 43], distributed storage systems [18, 79, 87], SDN analytics [80], high-performance computing applications [40, 41, 74], multi-robot systems [44], information retrieval systems [86, 87], and large-scale network simulations [10] (see full taxonomies in [45, 74]). One way to reconcile these empirical observations and Amdahl's law is the *scaled size model* [41]. In the *fixed size* model [41], the size of workers' sub-problems remains constant as we scale the system [42]. Under this assumption faster-than-linear scaling is impossible [25]. However, when this assumption is relaxed, say, when the workers' jobs get progressively smaller or execution gets gradually faster as we add more parallel workers ("scaled size model"), superlinear scaling often emerges [40, 43, 44, 80].

Conditions under which superlinear scaling emerges are widely discussed [41, 86, 87], analyzed [45, 74], and debated [25, 34, 35]. What is missing is a generic design methodology to optimize distributed systems towards faster-than-linear scaling, identifying when it is possible and when it is not. Our main contribution is a new optimization strategy to fill this gap.

3 Distributed self-adjusting systems

We now present the distributed self-adjusting systems architecture. Its two critical components are: a locality-boosting load balancer that increases the locality of reference in workers' input streams, and a self-adjusting algorithm that adaptively exploits that structure to process inputs more efficiently. The technique is a pure software optimization – no new cache space is required – though it subsumes distributed caching as a special case and benefits from fast memory when available.

3.1 Locality-boosting load balancing

The first component in our architecture is a locality-boosting load balancer that distributes work across parallel *workers* (servers, processors, or nodes) while improving the locality of reference in the input each worker receives [21].

Locality of reference is the property of a sequence of inputs that subsequent items are statistically dependent on each other. 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. 2).

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 items. 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

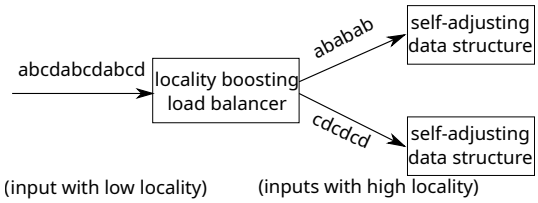


Figure 2 A locality boosting load balancer partitions the input sequence of a given locality into subsequences with higher locality. Self-adjusting data structures perform better on inputs with higher locality.

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 order packet batches flow-wise, thereby enabling more efficient processing [29, 56].

3.2 Self-adjusting algorithms

The second critical enabler is *self-adjusting algorithms*: data structures that automatically reorganize based on the input sequence to speed up future accesses to frequently used items. Self-adjustment improves performance only when the input exhibits sufficient locality; on uniform input it merely adds overhead. We review three prominent examples.

Caches. The textbook self-adjusting structure is a *cache* [75, 79, 94]: frequently accessed items are stored in fast memory, bypassing the full processing pipeline or slow backing store. Caches require no prior knowledge beyond a locality promise – when the promise holds they cheaply improve throughput; when there is no locality they add overhead. Caches do not have to be hardware: a key-value store can cache a slow database [26], and a kernel flow cache can accelerate a user-space switch [73].

List lookup. The *move-to-front (MTF) list* is a widely used self-adjusting data structure: after each access the requested item is moved to the front of the list, improving future lookup time at minimal cost. MTF is near-optimal even with full knowledge of future requests [83], handles both spatial and temporal locality, and adds overhead only on uniformly distributed input. Classic applications include information retrieval, compression [16], rule matching in OpenFlow and P4 switches [69], and packet classification (see later). Every list reorganization algorithm gives rise to a different cache management algorithm [83].

Search trees. A *splay tree* is a self-adjusting search tree that dynamically moves popular items closer to the root and less-accessed items toward the bottom [9, 17, 84]. Unlike self-balancing trees (red-black, AVL), splay trees reorganize with respect to the *queries posed* rather than the items stored, improving future access time on high-locality input. Splay trees are widely used in associative memory and data compression [49], and 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. 2) combined with a move-to-front list 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. Recall, uniformly distributed input is 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 on only m/k different items (see Fig. 2). 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, and another $k\times$ speedup as the total compute capacity available to the system grows 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):

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

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

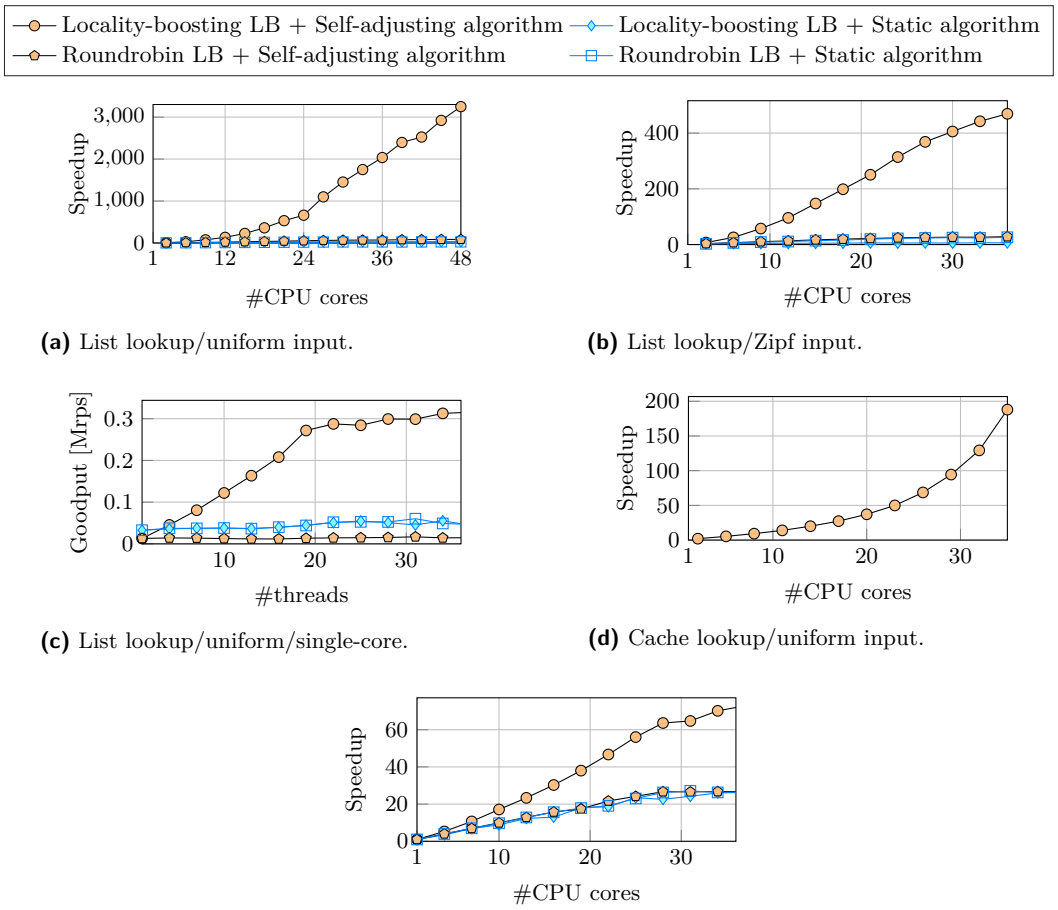
For small values of k, we obtain $O(k^2)$ scaling. As k grows sufficiently large, say, when $k = m$, the workers' input reduces to a singleton ($m/k = 1$). From this point the distributed MTF list reduces into a simple parallel hash table and superlinear speedup degrades into an "ordinary" Amdahl's scaling 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 jobs workers will have to process (recall the "scaled size model", § 2). Denote the "job size reduction" attainable with k workers by $\ell(k)$. Second, there is a "parallelizability" gain, denoted by $q(k)$, that is obtained by k self-adjusting workers processing the reduced workloads. In the Appendix we present a formal definition of these terms and define the below *scaling law for distributed self-adjusting systems*:

$$S(k) = \frac{T(1)}{T(k)} = \frac{1}{s + \frac{1-s}{q(k)\cdot\ell(k)}} \ . \tag{3}$$

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

Superlinear growth is always transient: eventually the system hits a bottleneck and falls back to (sub)linear scaling. Speedup is measured relative to the single-threaded non-self-adjusting baseline; see § 5 and Appendix A for feasibility conditions and a full formal characterization.



(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 3 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 ($m = 50k$, cache hit rate $\delta = 0.05$, $\rho = 100k$ 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 goodput (million requests per second).

3.4 Empirical evidence

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, written in Go, runs a configurable load balancer and k goroutine workers, each executing the selected lookup algorithm (static or MTF list; LRU cache [33]; balanced tree [31] or splay tree [32]) on a random input sequence. To keep the workload CPU-bound, tree comparisons cost w extra cycles and cache misses cost ρ cycles. Total execution time – including warm-up, request generation, and goroutine scheduling overhead – is measured with nanosecond precision. For the hardware platform, see § 4.3.

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

Figure 4 Sample firewall rule set. Source ports do not matter.

Fig. 3 shows the results. First, *the right combination robustly delivers superlinear speedup* regardless of problem domain or input distribution: on worst-case uniform input we obtain $3,300\times$ speedup for list access on 48 CPU cores ($70\times$ beyond linear), $200\times$ on LRU caches, and $65\times$ on tree search with 36 cores. Second, *only the combination of locality-boosting load balancing and self-adjusting algorithms produces superlinear speedup*; all other combinations (round robin with any algorithm, or any load balancer with a static algorithm) fall back to (sub)linear scaling. Third, *self-adjustment has overhead on a single thread* (Fig. 3c): the single-threaded self-adjusting algorithm is slower than the static one on uniform input; on skewed input (Zipf, Fig. 3b) it is 2–$2.5\times$ faster, but superlinear speedup still requires locality-boosting. Fourth, *superlinear gain appears even with constant CPU*: with an increasing number of threads sharing one core (Fig. 3c), the self-adjusting combination delivers linear speedup – only one of the two $k\times$ factors is active when total CPU is fixed.

4 Superlinear scaling in the Linux kernel

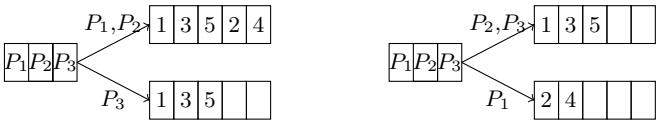
We demonstrate our methodology on software packet classification [39]. We chose this domain because `nftables` – the default Linux firewall – uses a static linked list for rule matching (a natural MTF candidate, though rule dependencies prevent naive MTF application, see § 4.1), classifiers are notoriously hard to cache [22], and the kernel provides hardware load balancers for the locality-boosting component [76]. Our goal is to verify that superlinear scaling can be robustly reproduced; producing the fastest possible classifier is a nongoal. Nonetheless, the re-engineered classifier will prove several times faster than the default Linux implementation. (A second case study on Memcached+PostgreSQL is in Appendix B.)

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. 4). 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` engine is a virtual machine that uses a Domain Specific Language for parsing and matching packet header fields [66]. 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.

A naive application of the MTF heuristic to `nftables` would break firewall semantics. Rules may *depend* on each other: rule u depends on rule v if they have overlapping match criteria, v has higher priority, and they define different actions; moving u before v causes



(a) Imperfect separation **(b)** Perfect separation

Figure 5 Locality-boosting load balancing over packet sequence P_1 (matching rule 4 of the sample classifier in Fig. 4) 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.

misclassification. The *Move-Recursively-Forward* (MRF) algorithm [1] resolves this by pushing an accessed rule forward only up to its nearest dependency, then recursively moving that dependency forward too. MRF reduces to plain MTF when there are no dependencies and degrades to a static list when all rules form a single dependency chain. In general, MRF moves frequently accessed rules – together with their dependencies – toward the front of the chain, improving lookup performance on high-locality input without breaking classifier semantics [1], and is near-optimal in the same competitive sense as MTF.

We implemented MRF on top of `nftables` with multiple parallel instances, each maintaining its own rule order in a private per-CPU pointer array over a shared static rule list, enabling lockless list reordering and rule addition/deletion. Rather than the recursive formulation of [1], we use an iterative implementation to avoid deep call stacks in the kernel: when moving a rule forward, we check for overlap with the preceding rule using a range-based representation extracted from the `nftables` bytecode; if they overlap we attempt to push the blocking dependency forward instead, otherwise the two independent rules are swapped. A more efficient approach would precompute dependencies at insertion/deletion time; we leave this for future work.

4.2 Locality-boosting load balancing

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* 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 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 their active rule sets (see an example in Fig. 5). Note that the same rule duplication problem plagues many software packet classifier algorithms [38, 57, 82, 90].

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 § 6). Therefore, we adopt a simple hash-based load balancing scheme here that implements only "imperfect

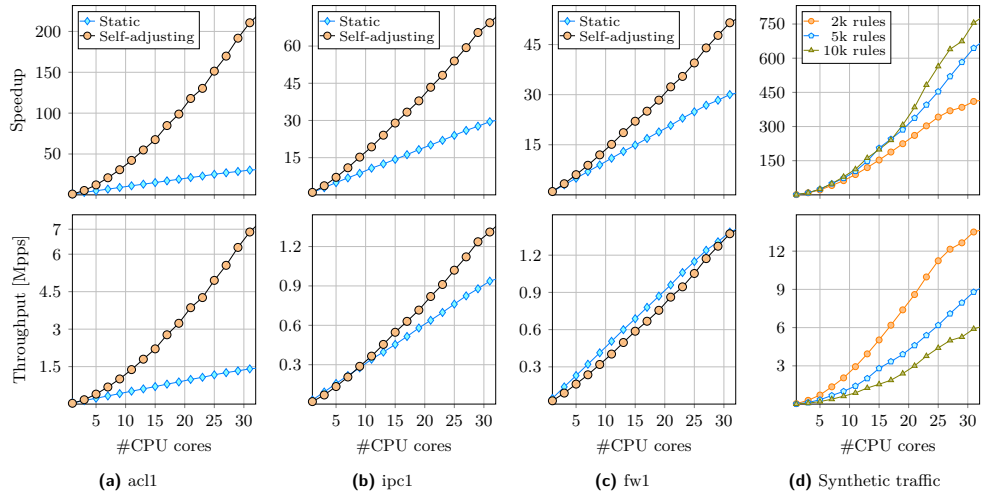


Figure 6 Macrobenchmarks: Scaling on 3 ClassBench rulesets generated from different seeds, containing 5000 rules each (panel (a), (b) and (c)), and synthetic rule set with uniform traffic and different rule sizes (panel (d)). Upper row shows relative speedup and the bottom row shows absolute throughput (packet rate in million packets per sec, mpps).

rule set partitioning". Our load balancer will however be fully implemented in hardware and run at line rate. This is crucial in order 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, [12, 76]) 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` [24], connected back-to-back to the SUT over Intel XL710 40GbE NICs. We used standard Ubuntu 22.04.4 LTS OS VMs with NIC-passthrough, running a patched v6.5 Linux kernel on the SUT replacing the `nftables` packet classifier with our own self-adjusting implementation. The benchmarks use the Tipsy network testing automation and visualization tool [60]. 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` [63,88], 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 [63, 64]. We also used a series of *synthetic rule sets* and matching packet traces for conducting controlled microbenchmarks. For each synthetic rule set we generated a matching 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.

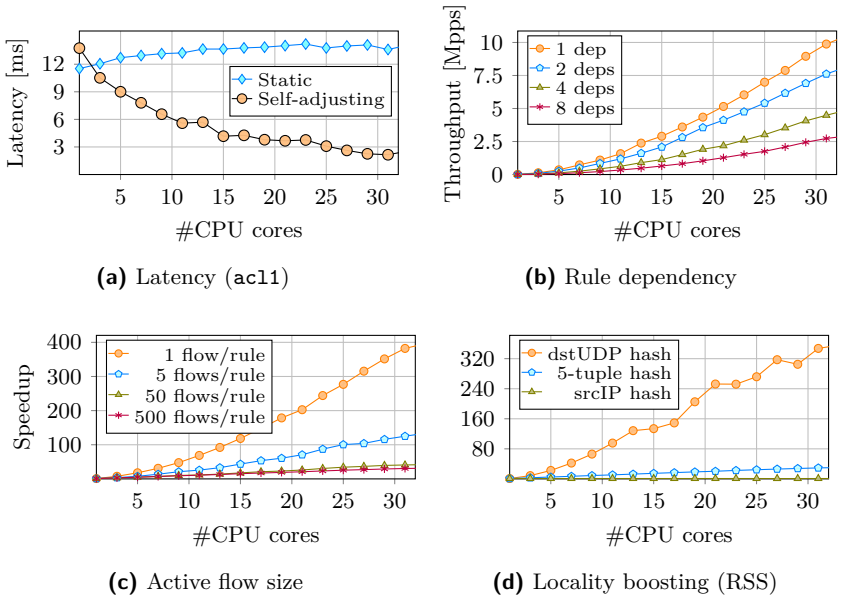
Macrobenchmarks. First, we ask whether superlinear scaling can be reproduced with real workloads. Fig. 6a, Fig. 6b and Fig. 6c give the speedup and the raw packet rate obtained with the default `nftables` packet classifier and our self-adjusting implementation on 3 ClassBench rule sets, each containing 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 shows 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. 6c. The absolute packet rate of 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 sufficient parallel resources, distributed self-adjustment eventually surpasses static algorithms even in terms of raw performance. In other words, when scaling is superlinear even a slow baseline becomes extremely fast ultimately.

Latency. The mean per-packet latency is shown in Fig. 7a. We observe that *superlinear speedup transforms into massive latency reduction*, resulting in 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. Next we turn to controlled microbenchmarks over synthetic input, which we fine-tune to highlight the effect of specific workload characteristics 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 § 4.2). We used the following template to generate synthetic rule-sets of configurable size:



(a) Latency (`acl1`) **(b)** Rule dependency

(c) Active flow size **(d)** Locality boosting (RSS)

Figure 7 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 (independent rules); and (d) speedup with different RSS hash functions (same rules).

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	ACCEPT
...	...	...	...	...	...

The source and destination 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 generated a matching packet trace containing one flow per rule.

Fig. 6d shows the results. The 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 completes the 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. Rule-dependencies have a crucial role in self-adjustment, since for every rule with nonzero dependencies not just the rule but all its dependencies will also become active, bloating the active rule sets. To measure the effects of rule dependencies we created 3 synthetic rule sets with increasingly long dependency chains using the below template:

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
...	...	...	...	...	...

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. Unfortunately, rule set size also increases d times, but this should not affect the basic superlinear speedup trends (recall Fig. 6d). 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. 7b 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 values 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 the reason for the slow scaling on the `fw1` ClassBench seed. We observed a similar slowdown when sending huge 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. We generated 4 traces containing 1, 5, 50, and 500 uniformly distributed flows per rule, respectively. The results in Fig. 7c 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 46–400× speedup on 32 cores. However, for 500 flows the superlinear trend disappears and scaling degrades to linear (32× speedup on 32 cores). We traced the issue 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.

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. 7d 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 all

together: 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 is 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. 7d, with roughly the same speedup as with no flow diversity in Fig. 7c. This confirms that faster-than-linear growth appears only if the load balancer is indeed "locality-boosting".

5 Limitations

Our experimental evaluations indicate that superlinear scaling occurs only under particular circumstances. In this section, we describe those conditions and share practical lessons from our own work, culminating in a set of design guidelines.

Optimize the load balancer for the worker implementation (or the other way around) so that the load balancer boosts exactly the type of locality workers can exploit. A load balancer that boosts temporal locality will not improve the performance of a worker designed for spatial locality (and *vice versa*). Similarly, a load balancer splitting traffic by hashing on the IP source-destination pair will work for an L3 classifier that filters only on the network layer header fields, but it may cease to be properly "partitioning" if the classifier considers transport layer header fields as well. Properly matching the load balancer to the self-adjusting worker remains a laborious manual task for the moment.

Make workers' internal data structures independent so that each worker can autonomously rearrange itself with respect the locality of its own input. Shared data structures will not work. For instance, [30] maintains a single cache to offload popular Memcached queries that is shared across kernel threads. This blocks parallel self-adjustment by (re)mixing the locality in the threads' input into a single unstructured workload for the shared cache. In contrast, our MRF classifier implementation carefully allocates the per-worker rule-lists as private per-CPU pointer arrays, so that each worker can maintain its own rule order on top of the shared static rule list.

Avoid sequential bottlenecks that may block parallel speedup prematurely. We identified this issue during the design of our self-adjusting packet classifier. In order to improve the locality-boosting property of the load balancer we experimented with running it in the Linux kernel's RPS function, which admits more flexible traffic splitting rules than plain hardware RSS. This would result in better locality boosting by letting us fine-tune the partitioning function for the classifier rules, in contrast to the hardware RSS that supports only hash-based load balancing. Despite that we identified substantial improvement at low packet rates, the single RPS kernel thread quickly posed a firm sequential bottleneck, blocking further scaling well before superlinear speedup could appear.

Make self-adjustment matter so that workers can take advantage of improved locality. If in a distributed caching system cached access is only moderately faster than uncached access then very little gain can be obtained from improving workers' cache efficiency. Likewise, if k

parallel caches are enough to cache the entire workload, adding a new worker will have little effect. For attaining super-linear speedup workers in a distributed self-adjusting system must be CPU-bound (so that adding parallel CPU resources matter) or, for caching in particular, memory-bound (so that additional fast cache resources will translate into faster parallel execution). This is clearly the case for the self-adjusting classifier, as evidenced by our benchmarks.

Choose an adequate baseline for measuring speedup. An inadequate benchmarking methodology may easily ruin a faster-than-linear scaling trend or, what is worse, show a spurious superlinear speedup [35]. In this paper we report speedup with respect to the single-core non-self-adjusting baseline (in line with (1)). Comparing to the self-adjusting baseline superlinear scaling often disappears. To get a full picture, analyze speedup and raw performance results side-by-side. This allows to spot cases of spurious speedup resulting from a slow baseline.

6 Related work

Superlinear scaling. Amdahl's famous scaling law [4], asserting sublinear speedup and diminishing returns for parallelization, is a cornerstone result in distributed computing [14, 37, 42, 42, 54]. During the almost 60 years since its first publication many useful extensions of the basic law were published [20, 36, 37, 47, 62, 70]. Remarkably, faster-than-linear scaling trends were observed in a broad range of production workloads [10, 26, 40, 41, 43, 44, 74, 80, 86, 87]. For instance, [43] shows superlinear speedup for PostgreSQL and attributes this to a new "cache plan" for caching compiled SQL queries at each thread, [74] 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 [7]. Meanwhile, there have been heated debates on the controversies related to superlinear scaling: 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 [25].

There seem to be two common strategies to obtain superlinear scaling [45, 74]: either do disproportionately less work per worker as the system is scaled (scaled size model [74]), or add more resources per thread [45]. These techniques, however, are difficult to apply beyond specific use cases [41] or require adding more cache space [74]. One way to realize the scaled size model in practical systems is choosing a proper load dispatching strategy that matches the worker implementation. Indeed, there have been several observations that the proper load balancing strategy can improve data locality, and hence cache efficiency, at workers, yielding measurable speedup [21]. To the best of our knowledge, ours is the first methodology to extend this observation from caching systems to general compute-bound workloads: by observing that caching is only a special case of self-adjustment, we show that the proper combination of locality-boosting load balancing and self-adjusting algorithms can reproduce superlinear growth in several applications.

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 [81], 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 [76] and RSS++ is a dynamic

receive side scaling mechanism aiming to keep CPU load constant [12]. 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 [29, 56], and SAX-PAC can be used to decompose a classifier rule set with many dependencies into multiple smaller but independent rule sets [53]. Hicuts [38], Hypercuts [82], Efficuts [90], and CutSplit [57] define "intelligent" packet header space cuts [52] to partition a rule set along a decision tree into smaller rule lists stored in the leaves of the tree. These schemes are complementary to our approach: while [38, 57, 82, 90] 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 [55], database accelerators [26, 30, 67], distributed web caching and CDNs [94], and microservices [93]; move-to-front (MTF) lists are used for computing point maxima and convex hulls [15], program compilation and interpretation [46], detecting collisions in hash tables [46], and data compression [16]; further examples are splay trees [84], self-adjusting skip lists [17], push-down trees [9], or self-adjusting geometric data stores [72], etc. Another example for self-adjustment are runtime optimization frameworks which can just-in-time recompile code to specialize it to a particular structured input [29, 56, 59, 65]. 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.

7 Conclusions

In this paper, we provide theoretical and empirical evidence that combining locality-boosting load balancing with parallel self-adjusting algorithms can yield faster-than-linear speedup in certain applications and under specific workload conditions. Our main contribution is the identification of the key design pattern that consistently appears in use cases where superlinear scaling emerges, along with a characterization of the conditions required for it to occur. Through extensive simulations, we demonstrate that this optimization technique can achieve scaling significantly beyond what has previously been observed, and we further illustrate its applicability on several widely used production applications. 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. We also reproduce faster-than-linear scaling on a Memcached+PostgreSQL distributed storage system. 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 [48] or explainable AI inferencing seem like appealing application candidates. In addition, identifying other reusable design patterns that enable superlinear scaling would be a valuable direction.

References

1 Vamsi Addanki, Maciej Pacut, Arash Pourdamghani, Gabor Rétvári, Stefan Schmid, and Juan Vanerio. Self-adjusting partially ordered lists. In *IEEE Conference on Computer Communications*, IEEE INFOCOM, pages 1–10, 2023.

2 Susanne Albers and Sonja Lauer. On list update with locality of reference. *J. Comput. Syst. Sci.*, 82(5):627–653, 2016. doi:10.1016/J.JCSS.2015.11.005.

3 Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7):922–927, 2005. doi:10.1109/TC.2005.119.

4 Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. Association for Computing Machinery, 1967. doi:10.1145/1465482.1465560.

5 Gunnar Andersson. An approximation algorithm for max p-section. In *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science*, pages 237–247, 1999. doi:10.1007/3-540-49116-3_22.

6 Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006. doi:10.1007/S00224-006-1350-7.

7 Samuel Arbesman, Jon M. Kleinberg, and Steven H. Strogatz. Superlinear scaling for innovation in cities. *Phys. Rev. E*, 79, 2009.

8 Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic balanced graph partitioning. *SIAM J. Discret. Math.*, 34(3):1791–1812, 2020. doi:10.1137/17M1158513.

9 Chen Avin, Kaushik Mondal, and Stefan 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. doi:10.1007/978-3-030-61792-9_12.

10 Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. Unison: a parallel-efficient and user-transparent network simulation kernel. In *European Conference on Computer Systems*, EuroSys, pages 115–131, 2024. doi:10.1145/3627703.3629574.

11 Amotz Bar-Noy and Michael Lampis. Online maximum directed cut. *J. Comb. Optim.*, 24(1):52–64, 2012. doi:10.1007/S10878-010-9318-6.

12 Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. RSS++: load and state-aware receive side scaling. In *International Conference on Emerging Networking Experiments And Technologies*, ACM CoNEXT '19, pages 318–333, 2019.

13 Tom Barbette, Erfan Wu, Dejan Kostić, Gerald Q. Maguire, Panagiotis Papadimitratos, and Marco Chiesa. Cheetah: a high-speed programmable load-balancer framework with guaranteed per-connection-consistency. *IEEE/ACM Transactions on Networking*, 30(1):354–367, 2022. doi:10.1109/TNET.2021.3113370.

14 G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *Computer*, 39(1):110–112, 2006. doi:10.1109/MC.2006.29.

15 Jon Louis Bentley, Kenneth L. Clarkson, and David B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica*, 9(2):168–183, 1993. doi:10.1007/BF01188711.

16 Jon Louis Bentley, Daniel Dominic Sleator, Robert Endre Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986. doi:10.1145/5684.5688.

17 Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 1106–1114, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347203.

18 Daniel Brahneborg, Wasif Afzal, Adnan Čaušević, and Mats Björkman. Superlinear and bandwidth friendly geo-replication for store-and-forward systems. In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT)*, pages 328–338. SciTePress, 2020. doi:10.5220/0009835403280338.

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

20 Kirk W. Cameron and Rong Ge. Generalizing Amdahl's Law for power and energy. *Computer*, 45(3):75–77, 2012. doi:10.1109/MC.2012.92.

21 Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 249–259. Association for Computing Machinery, 1993. doi:10.1145/155332.155358.

22 F. Chang, Wu chang Feng, and Kang Li. Approximate caches for packet classification. In *IEEE INFOCOM*, volume 4, pages 2196–2207, 2004.

23 Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan 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, March 2016. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud.

24 Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, October 2015.

25 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. doi:10.1016/0167-8191(86)90024-4.

26 Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, 2004.

27 Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73. USENIX Association, 2024. URL: https://www.usenix.org/conference/nsdi24/presentation/fried.

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

29 Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan 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. URL: https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni.

30 Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles 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. URL: https://www.usenix.org/conference/nsdi21/presentation/ghigoff.

31 hashicorp/btree. https://pkg.go.dev/github.com/google/btree.

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

33 hashicorp/golang-lru. https://pkg.go.dev/github.com/hashicorp/golang-lru/v2.

34 Neil Gunther. Superlinear scalability. In *Symposium and Bootcamp on the Science of Security*, HotSoS '13, 2013.

35 Neil Gunther, Paul Puglia, and Kristofer Tomasette. Hadoop superlinear scalability: The perpetual motion of parallel performance. *Queue*, 13(5):20–42, 2015.

36 Neil J. Gunther. A general theory of computational scalability based on rational functions, 2008. arXiv:0808.1431.

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

38 P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000. doi:10.1109/40.820051.

39 Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001. doi:10.1109/65.912717.

40 Marjan Gusev and Sasko Ristov. Superlinear speedup in Windows Azure cloud. In *IEEE International Conference on Cloud Networking (CLOUDNET)*, pages 173–175, 2012. doi:10.1109/CLOUDNET.2012.6483679.

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

42 John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988. doi:10.1145/42411.42415.

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

44 Heiko 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. doi:10.1007/978-3-319-77610-1_3.

45 D.P. Helmbold and C.E. McDowell. Modelling speedup (n) greater than n. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):250–256, 1990. doi:10.1109/71.80148.

46 James H. Hester and Daniel S. Hirschberg. Self-organizing linear search. *ACM Comput. Surv.*, 17(3):295–311, 1985. doi:10.1145/5505.5507.

47 Mark D. Hill and Michael R. Marty. Amdahl's Law in the multicore era. *Computer*, 41(7):33–38, 2008. doi:10.1109/MC.2008.209.

48 Haiyang Jiang, Guangxing Zhang, Gaogang Xie, Kavé Salamatian, and Laurent 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, pages 137–146, 2013. doi:10.1109/ANCS.2013.6665196.

49 Douglas W. Jones. Application of splay trees to data compression. *Communications of the ACM*, 31(8):996–1007, 1988. doi:10.1145/63030.63036.

50 Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 97–112, 2017. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan.

51 Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire 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, April 2018. URL: https://www.usenix.org/conference/nsdi18/presentation/katsikas.

52 Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Symposium on Networked Systems Design and Implementation*, USENIX NSDI, pages 113–126, 2012. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

53 Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. SAX-PAC (Scalable And EXpressive PAcket Classification). In *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pages 15–26, 2014. doi:10.1145/2619239.2626294.

54 S. Krishnaprasad. Uses and abuses of Amdahl's Law. *J. Comput. Sci. Coll.*, 17(2):288–293, December 2001. doi:10.5555/775339.775386.

55 Jason Lei and Vishal Shrivastav. Seer: Enabling Future-Aware online caching in networked systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 635–649, 2024. URL: https://www.usenix.org/conference/nsdi24/presentation/lei.

56 Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor 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.

57 Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM*, pages 2645–2653, 2018. doi:10.1109/INFOCOM.2018.8485947.

58 Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael 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, April 2014. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

59 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. doi:10.1109/JPROC.2019.2896848.

60 Tamás Lévai, Gergely Pongrácz, Péter Megyesi, Péter Vörös, Sándor Laki, Felicián Németh, and Gábor 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. doi:10.1109/JSAC.2018.2871307.

61 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.

62 Ami 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. doi:10.1109/ISPA.2012.47.

63 Jiří Matoušek, Gianni Antichi, Adam Lučanský, Andrew W. Moore, and Jan Kořenek. ClassBench-ng: recasting ClassBench after a decade of network evolution. In *Symposium on Architectures for Networking and Communications Systems*, IEEE/ACM ANCS, pages 204–216, 2017.

64 Sebastiano Miano. sebymiano/pcap-utils. https://github.com/sebymiano/pcap-utils.

65 Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni 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, pages 1148–1164, 2022. doi:10.1145/3503222.3507769.

66 The nftables project. https://wiki.nftables.org.

67 Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, April 2013. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

68 Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, 2018.

69 ONF. Openflow reference release. https://github.com/mininet/openflow, 2013.

70 I. Onyuksel and S.H. Hosseini. Amdahl's law: a generalization under processor failures. *IEEE Transactions on Reliability*, 44(3):455–462, 1995. doi:10.1109/24.406581.

71 Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for NFV applications. In *Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, 2015. doi:10.1145/2815400.2815423.

72 Eunhui Park and David M. Mount. A self-adjusting data structure for multidimensional point sets. In *Algorithms - ESA Annual European Symposium*, volume 7501, pages 778–789, 2012. doi:10.1007/978-3-642-33090-2_67.

73 Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015. URL: `https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff`.

74 Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. Superlinear speedup in HPC systems: Why and when? In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898, 2016. `doi:10.15439/2016F498`.

75 Ori Rottenstreich and János Tapolcai. Optimal rule caching and lossy compression for longest prefix matching. *IEEE/ACM Transactions on Networking*, 25(2):864–878, 2016. `doi:10.1109/TNET.2016.2611482`.

76 Scaling in the linux networking stack. `https://www.kernel.org/doc/Documentation/networking/scaling.txt`.

77 Packet classification and access control. `https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html`.

78 Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter 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. URL: `https://www.usenix.org/conference/nsdi21/presentation/sapio`.

79 Chris Sears. The elements of cache programming style. In *4th Annual Linux Showcase & Conference (ALS 2000)*. USENIX Association, 2000. URL: `https://www.usenix.org/conference/als-2000/elements-cache-programming-style`.

80 Sdn analytics and control: Superlinear. `https://blog.sflow.com/2010/09/superlinear.html`, 2010.

81 Justine Sherry. The I/O driven server: From SmartNICs to data movement controllers. *Computer Communications Review (CCR)*, 53(3), 2023. `doi:10.1145/3649171.3649174`.

82 Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 213–224, 2003. `doi:10.1145/863955.863980`.

83 Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985. `doi:10.1145/2786.2793`.

84 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985. `doi:10.1145/3828.3835`.

85 Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. In *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 43–56, 2017. `doi:10.1145/3098822.3098826`.

86 Herb Sutter. Going superlinear. Dr. Dobb's J., 2008. URL: `https://www.drdobbs.com/cpp/going-superlinear/206100542`.

87 Herb Sutter. Super linearity and the bigger machine. Dr. Dobb's J., 2008. URL: `https://www.drdobbs.com/parallel/super-linearity-and-the-bigger-machine/206903306`.

88 David E. Taylor and Jonathan S. Turner. ClassBench: a packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3):499–511, 2007. `doi:10.1145/1295237.1295239`.

89 William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open VSwitch Dataplane ten years later. In *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '21, pages 245–257, 2021. `doi:10.1145/3452296.3472914`.

90 Balajee Vamanan, Gwendolyn 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, pages 207–218, 2010. `doi:10.1145/1851182.1851208`.

91 Wikipedia. Speedup: Super-linear speedup. `https://en.wikipedia.org/wiki/Speedup#Super-linear_speedup`.

92 Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 299–312, 2018. URL: `https://www.usenix.org/conference/nsdi18/presentation/woo`.

93 Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. MuCache: A general framework for caching in microservice graphs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 221–238, 2024. URL: `https://www.usenix.org/conference/nsdi24/presentation/zhang-haoran`.

94 Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. 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 Analysis

Scaling in distributed systems refers to the improvement in performance achieved by adding more machines to a system. It is 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).

A.1 The model

A.1.1 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, ignoring 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).

A.1.2 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

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

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.

A.2 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 $q(k)$, $q(k) \leq k$.

A.3 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.

A.3.1 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

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

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 [83] captures more general data structures than just lists and caching under a common characterization of g_D.

A.3.2 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 $f_k : \mathcal{U} \to \{1, 2, \ldots, k\}$ 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

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

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.

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

Note the total work decreases by r for the stream σ.

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

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

A.4 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

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

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[1]. 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.

A.5 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 3.** *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 $f_k : \mathcal{U} \to \{1, 2, \ldots, k\}$ 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 is characterized in two dimensions: load balancer efficiency $\ell(k)$ and parallelizability $q(k)$. This gives us speedup

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

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.

[1] 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 [12].

A.6 Impossibility of scaling for self-adjusting distributed systems (a negative result)

We conclude by outlining a scaling impossibility law, analogous to Amdahl's law for self-adjusting distributed systems. We conclude that for any stream σ, scaling is limited to the initial phase and cannot continue indefinitely with k. Therefore, the superlinear scaling observed in practice is only merely a transient phenomenon.

▶ **Theorem 4.** *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.

Two key consequences arise from the above theorem.

1. **Superlinearity is an initial-only phenomenon**
   For any input sequence σ, the maximum achievable multiplicative $\ell(k)$ is fixed. This occurs when k is large enough so the load balancer f_k isolates all working sets. However, 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 scale with the parameter k due to additional resources, but the savings from load balancing do not grow indefinitely with k.

   ▶ **Observation 5.** *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 $q \leq k$, this implies that superlinear scaling cannot continue indefinitely with k.*

2. **Tight analysis for LRU caches and MTF lists**
   In case of 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, the analysis of scaling is tight for these algorithms.
   In particular, the LRU algorithm for caching cannot scale superlinearly indefinitely. Therefore, the scaling observed in the literature is only an initial effect caused by the combined influence of the increased number of machines and the load balancer.

A.7 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,

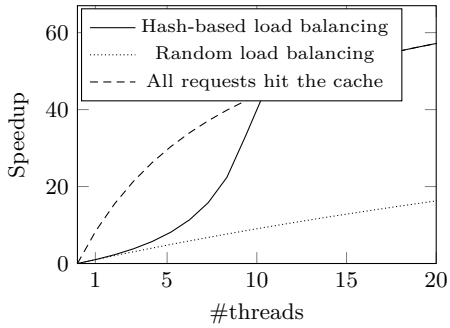


Figure 8 Scaling laws for distributed caching.

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 [28, 61]. If the cuts are balanced (a natural choice), then the problem is known as *maximum k-section* [5] (from the perspective of maximizing the cut) and *minimum graph k-balanced partitioning* [6] (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 [5], and in the latter model, we have a polylogarithmic approximation [6].

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 [11] and online graph partitioning [8].

B Superlinear scaling in distributed caching

Superlinear scaling often emerges in systems where a "fast" *distributed cache* is deployed in front of "slow" processing system or storage engine [43, 80, 87]. Examples include multi-processor CPUs with unshared Level-1 fast cache memory that make access to program arguments more efficient [74], runtimes that selectively "memoize" the results of costly computations [3], FIB caches in OS network stacks that maintain the most recent IP routes in fast memory to sidestep longest prefix matching [75], hierarchical (mega)flow caches that serve as a fast-path in programmable software switches [73], etc. All these workloads may benefit from caches becoming more efficient as the system is scaled and, potentially, show superlinear speedup on certain workloads. We use Memcached as a fast cache for PostgreSQL [26, 30, 58, 67].

We quantify this with 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. The single-worker cache hit rate 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 $\frac{c}{m/k} = k\delta$ ($k\delta \leq 1$). This puts the lookup time of the system of k parallel caches to

$$T_c(k) = \begin{cases} s + \frac{1-s}{k}(k\delta + (1-k\delta)\rho) & \text{if } k\delta \leq 1 \\ s + \frac{(1-s)}{k} & \text{otherwise} \end{cases}, \tag{4}$$

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

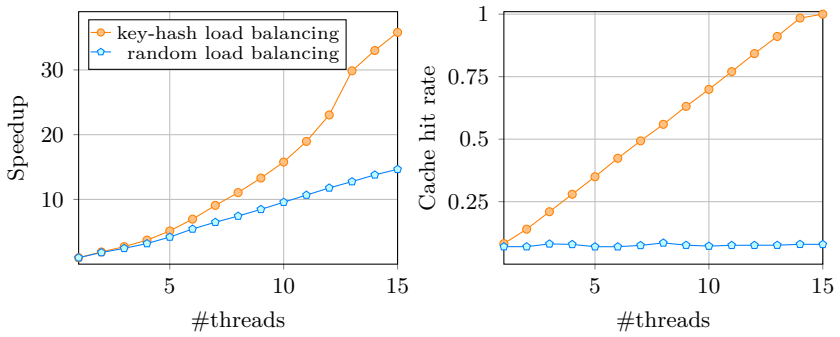


Figure 9 Results for a joint scaling of Memcached+PostgreSQL with and without key-hashing: speedup and cache-hit rate.

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. 8. 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 our case study Memcached is gradually scaled from a single replica to 15 replicas, playing the role of a distributed cache for a "slow" PostgreSQL v14 database. PostgreSQL is scaled proportionally to the number of Memcached replicas; in particular we run 4 PostgreSQL client threads per cache replica. We wrote a custom client performing cache-aside reads: on a miss, it reads from PostgreSQL and writes the result back to the cache. We used two different load balancing schemes to route key requests: random load balancing reads from a random Memcached replica, while key-hashing always reads/writes the same key from/to the same Memcached replica.

Fig. 9 shows the results for PostgreSQL pre-filled with 1,000,000 key-value pairs of 16 byte keys and 48 byte values, with a configurable number of PostgreSQL threads and Memcached replicas with 4 MB of cache each. As expected, superlinear scaling emerges with key-hashing, yielding 35× speedup with 15 Memcached replicas, 2.3× higher than linear scaling. In contrast, random request routing exhibits only linear scaling. The reason is the improving cache hit rate as Memcached is scaled: with 14 replicas we reach close to 100% cache hit rate and speedup falls back into the linear range, as predicted by the analysis.

We note that superlinear scaling in this context is extremely sensitive to certain benchmark parameters, like the number of Memcached replicas, PostgreSQL threads, and client threads. This is because for faster-than-linear scaling to appear Memcached replicas must be both CPU-bound (so that adding more replicas will improve throughput) *and* memory-bounded (so that improving cache hit rate will cause speedup) at the same time.