# NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres

Luo Mai† Lukas Rupprecht† Abdul Alim†
Paolo Costa‡ Matteo Migliavacca* Peter Pietzuch† Alexander L. Wolf†
†Imperial College London ‡Microsoft Research *University of Kent

## Abstract

Data centre applications for batch processing (e.g. map/reduce frameworks) and online services (e.g. search engines) scale by distributing data and computation across many servers. They typically follow a *partition/aggregation* pattern: tasks are first partitioned across servers that process data locally, and then those partial results are aggregated. This data aggregation step, however, shifts the performance bottleneck to the network, which typically struggles to support many-to-few, high-bandwidth traffic between servers.

Instead of performing data aggregation at edge servers, we show that it can be done more efficiently along network paths. We describe NETAGG, a software platform that supports *on-path aggregation* for network-bound partition/aggregation applications. NETAGG exploits a middlebox-like design, in which dedicated servers (*agg boxes*) are connected by high-bandwidth links to network switches. Agg boxes execute aggregation functions provided by applications, which alleviates network hotspots because only a fraction of the incoming traffic is forwarded at each hop. NETAGG requires only minimal application changes: it uses *shim layers* on edge servers to redirect application traffic transparently to the agg boxes. Our experimental results show that NETAGG improves substantially the throughput of two sample applications, the Solr distributed search engine and the Hadoop batch processing framework. Its design allows for incremental deployment in existing data centres and incurs only a modest investment cost.

**Categories and Subject Descriptors:** C.2.1 [Network Architecture and Design]: Network communications; Distributed Networks

**General Terms:** Design, Performance.

**Keywords:** Data Centres, Middleboxes, In-network Processing, On-path Aggregation.

## 1. INTRODUCTION

Many applications in data centres (DCs) achieve horizontal scalability by adopting a *partition/aggregation* pattern [4]. These include search [36] and query processing [44], dataflow computing [17, 25], graph [42] and stream processing [16, 54], and deep learning frameworks [24]. In the partition step, a job or request is divided into independent subtasks, which are executed in parallel by different work servers ("workers"). Each worker operates on a subset of the data and locally generates partial results. In the aggregation step, partial results are collected and aggregated to obtain a final result.

These applications are challenging from a network perspective because they rely on an aggregation step in which a large number of workers cause *many-to-few* traffic patterns among servers. For example, traces from Facebook's DCs show that 46% of the overall network traffic is generated in the aggregation phase [19]. This creates network bottlenecks for the following two reasons: (i) the *scarce inbound bandwidth* available at the edge servers (e.g. 1 Gbps or 10 Gbps) caps the maximum transfer rate; and (ii) DC networks typically exhibit some degree of *bandwidth over-subscription* [13, 29], limiting available inter-rack bandwidth.

As a result, the network is often cited as one of the main performance bottlenecks for partition/aggregation applications [4,20,36]. Interactive applications such as search are significantly impacted by network activity: the network, on average, contributes 12% of latency in Microsoft's Bing search engine, accounting for 34% of outliers and 21% of timeouts [36]. In Facebook map/reduce jobs, network transfers on average are responsible for 33% of the execution time of the jobs with a reduce phase, and in 16% of these jobs network transfers account for more than 70% of the execution time [20]; in Microsoft Scope jobs, the network is responsible for a 62% median increase in the reduce and shuffle time [8].

Existing approaches attempt to counter this problem by overprovisioning DC networks using full-bisection bandwidth topologies [2, 29, 30, 53], using specialised network technologies [31, 47, 61], or carefully scheduling data movements to avoid network hotspots [3, 20]. Fundamentally, however, all these approaches do not reduce the network traffic and, hence, ultimately their performance is limited by the scarce bandwidth at the end hosts. While many DCs are in the process of upgrading to 10 Gbps networks, cost-effective 40 Gbps networks without an increased oversubscription ratio are still years away.

In contrast, we propose to reduce network traffic by aggregating data along the network paths in a distributed fashion. We describe **NETAGG**, a **software middlebox platform** that provides an **on-path aggregation service**. Middleboxes have been used extensively in DCs to enhance network functionality [37, 48]. A middlebox is a network appliance attached to a switch that provides services such as firewalls, web proxies, SSL offloading, and load balancing [51]. To maximise performance, middleboxes are often implemented in hardware [45], and adopt a vertically integrated architecture focusing on a narrow function, e.g. processing standard packet headers or performing relatively simple payload inspection.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*CoNEXT'14,* December 02-05 2014, Sydney, Australia
Copyright 2014 ACM 978-1-4503-3279-8/14/12 ...$15.00
http://dx.doi.org/10.1145/2674005.2674996.

The need to reduce costs, shorten update cycles, and allow more rapid innovation have instead motivated several proposals for *software* middlebox platforms [9, 14, 50, 52], typically based on commodity server hardware. Advances in leveraging multi-core parallelism and kernel engineering allow such middlebox platforms to run competitively at or near line rate [43].

We leverage this trend by using middleboxes to execute *application-specific* aggregation functions at each hop. This exploits the observation that often the aggregation phase exhibits high data reduction [18, 25]. By virtue of this aggregation performed by middleboxes, the amount of network traffic is reduced at each hop, thereby loosening network bottlenecks and, ultimately, increasing application performance. This differs from traditional approaches for traffic redundancy elimination [6], which operate only at the network level and have only limited visibility (if any) into application semantics.

To process data at the highest possible rate, NETAGG aggregation middleboxes (or *agg boxes*) decompose aggregation computation into cooperatively scheduled aggregation tasks, which are executed in parallel across many CPU cores. Multiple agg boxes can exist in a topology to cooperatively form an *aggregation tree*. NETAGG uses *shim layers* at edge servers to intercept application traffic and redirect it transparently to agg boxes. This minimises the changes required to existing applications for benefitting from on-path aggregation.

Multiple applications with different requirements can share a NETAGG deployment because agg boxes manage the scheduling of aggregation tasks: e.g. they can give priority to aggregation computation on behalf of latency-sensitive online applications over throughput-oriented batch applications.

We evaluate NETAGG both at scale, using simulation, and on a 34-server testbed with two deployed applications: Apache Solr [11], a distributed search engine, and the Apache Hadoop map/reduce framework [10]. In simulation, we show that NETAGG reduces the flow completion time up to 88% compared to existing solutions. We also show that our NETAGG deployment improves the performance of Solr search queries by up to 9.3× and Hadoop jobs by up to 5.2×.

# 2. DISTRIBUTED DATA AGGREGATION

We now describe how application performance can become network-bound due to a partition/aggregation communication pattern in DCs (§2.1) and discuss previous attempts to solve this problem (§2.2). We then sketch how we make use of middleboxes to perform on-path aggregation (§2.3) and present the results of a simulation-based feasibility study that supports our claim of the effectiveness of the approach, both in terms of performance and cost (§2.4).

## 2.1 Partition/aggregation applications

The partition/aggregation pattern is at the core of many distributed DC applications. A partition/aggregation application has a set of *worker nodes* ("workers"), deployed on edge servers. In the *partition* step, a request or job is divided into independent subtasks, which are executed in parallel by different workers. Each worker operates on a subset of the data and locally generates partial results. In the *aggregation* step, partial results are collected by a *master node*, such as a frontend server, and aggregated into the final result.

For example, online search queries are sent to multiple index servers in parallel, each hosting a small portion of the web index. Each index server processes the query locally and returns the top $k$ results best matching the query. These partial results are aggregated to select the final set of results returned to the user.

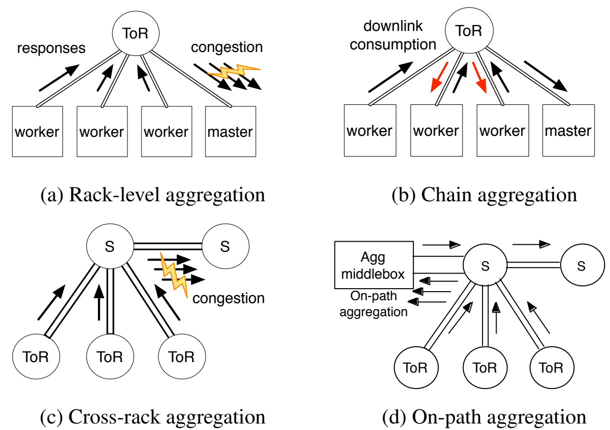


Figure 1: Distributed data aggregation strategies

Similarly, in map/reduce, the input data is partitioned into small chunks (with a typical size of 64–128 MB) and processed by a number of parallel map tasks (the *map* phase). The intermediate results are then sent to one or many reduce tasks (the *shuffle* phase), which perform the final step of aggregation (the *reduce* phase). Other frameworks, such as graph [42] or stream processing systems [16], adopt a similar approach for scaling.

Typically, the aggregation functions used in these applications exhibit two properties. First, they are *associative* and *commutative* [59], which implies that the aggregation step can be performed through a sequence of *partial* aggregations without affecting the correctness of the final result. Examples of aggregation functions exhibiting this property include `max`, `sum` and `top-k`. While there are some functions that do not satisfy this property, e.g. `median`, it does hold for many aggregations, especially those used for analytic queries [44] and graph processing [59].

Second, in most cases, the size of the aggregated results is a small fraction of the intermediate data generated. For example, the average final output data size in Google jobs is 40% of the intermediate data sizes [25]. In Facebook and Yahoo jobs, the reduction in size is even more pronounced: in 82% of the Facebook jobs with a reduce phase, the final output size is only 5% of the intermediate size, while for Yahoo jobs the number is 8% in 91% of the jobs [18]. Similar trends also hold for traces collected in Microsoft Scope, which show a reduction factor of up to two orders of magnitude between the intermediate and final data [12].

These two properties are important because they show that, by performing partial aggregation on-path, it is possible to reduce the traffic at each hop significantly, thus alleviating network bottlenecks as described below.

## 2.2 Edge-based aggregation

We are not the first to notice the benefits of partial aggregation of intermediate data to reduce network traffic. Prior work on interactive services [44] and dataflow frameworks [40, 59] proposes strategies to reduce inter-rack traffic through *rack-level aggregation*: one server per rack acts as an *aggregator* and receives all intermediate data from the workers in the same rack. The chosen server aggregates the data and sends it to another server for final aggregation (e.g. a map/reduce reducer).

The main drawback of rack-level aggregation is that its performance is limited by the inbound bandwidth of the aggrega-

tor (Figure 1a). For example, assuming 40 servers per rack and 1 Gbps edge network links, the maximum transmission rate per worker is only approximately 25 Mbps.

Rack-level aggregation can be extended to generalised *edge-based aggregation* by forming a $d$-ary tree of servers that aggregate data within a rack first and then progressively across racks. Intuitively, a lower $d$ (including the degenerate case of $d=1$, when the tree becomes a *chain*), leads to a higher maximum transmission rate per worker.

While $d$-ary trees eliminate some of the shortcomings of rack-level aggregation, they introduce new challenges: (i) small values of $d$ increase the depth of the tree, affecting the performance of latency-sensitive applications; (ii) small values of $d$ also increase intra-rack bandwidth usage because the incoming links of workers, as opposed to only outgoing links, are used to move data (Figure 1b). As we show in §4, this can drastically reduce the performance of other flows in the network that cannot be aggregated. For example, in a map/reduce job, only the shuffle flows can be aggregated, while other flows, e.g. used to read from the distributed file system, cannot. By using more links, the bandwidth available to these other flows is reduced.

More generally, a fundamental drawback of any edge-based aggregation approach is that it only applies to *intra-rack* traffic and not traffic in the core of the network. As shown in Figure 1c, if aggregation computation spans multiple racks, the links between aggregation switches can become a bottleneck, especially in the presence of over-subscription.

## 2.3 On-path aggregation with middleboxes

In contrast to edge-based aggregation approaches, we propose to use software middleboxes ("agg boxes") to perform aggregation along the network path. This allows us to minimise edge server bandwidth usage as well as congestion in the core of the network. While aggregation has been shown to increase efficiency in wireless sensor networks [41] and in overlay networks [15, 35, 58], its potential to improve the performance of partition/aggregation applications in DC networks remains unexplored. Existing work on data aggregation has either focused on per-packet aggregation, as in sensor networks, which is not appropriate for application-layer data flows in DCs, or on scalability issues for Internet system monitoring rather than on throughput requirements.

We assume a set-up similar to the one depicted in Figure 1d, in which middleboxes are directly attached to network switches and perform on-path aggregation. We create a spanning tree (hereafter referred to as the *aggregation tree*), in which the root is the final master node, the leaves are the workers and the internal nodes are the agg boxes. Each agg box aggregates the data coming from its children and sends it downstream. To fully exploit the path diversity available in most DC topologies, we use *multiple* aggregation trees that balance the traffic between them. We detail the design of our NETAGG middlebox platform in §3.

Due to its compatibility with existing middlebox deployments in DCs, our approach based on application-specific middleboxes can be deployed with low effort. It also allows for an incremental roll-out in which the agg boxes are attached to only a subset of the switches.

## 2.4 Feasibility study

A potential issue in using middleboxes based on commodity servers for on-path aggregation is the lower processing rate that they can achieve as compared to custom hardware solutions. To understand the feasibility of a software-only approach, we conduct a number of simulation experiments to understand (i) the minimum processing rate $R$ of an agg box required to achieve noticeable benefits and (ii) the performance/cost trade-off involved. We describe the simulation set-up and the full set of results in §4.1. Here we highlight the main results, which led to the NETAGG design described in the next section.

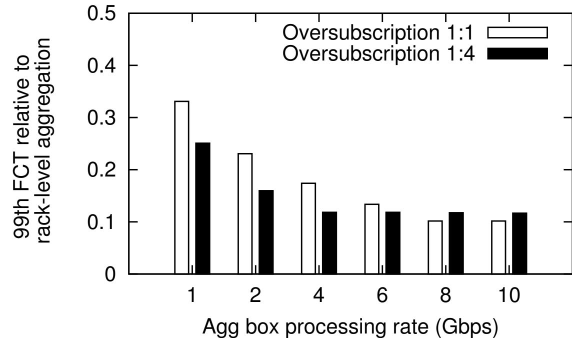


Figure 2: Flow Completion Time (FCT) for different aggregation processing rates $R$

**Simulation set-up.** We consider a three-tier, multi-rooted network topology, modelled after recently proposed scalable DC architectures [2, 29]. Each server is connected through a 1 Gbps link to a top-of-the-rack (ToR) switch. In this experiment, we assume an over-subscription ratio of 1:4, which is consistent with values reported in the literature [29]. We also show the results for the non-oversubscribed case. We consider a mid-sized DC with 1,024 servers and a synthetic workload modelled after Facebook network traces [19], in which 46% of traffic is aggregatable (see §4.1 for more details).

**Performance requirements.** As our baseline, we consider rack-level aggregation, as described above, and we assume ECMP [32] as the routing protocol. We use *flow completion time* (FCT), i.e. the time elapsed between the start and the end of a flow, as our metric [27] and vary the maximum processing rate that can be sustained by the agg boxes.

Figure 2 shows that relatively modest processing rates are sufficient to achieve significant benefits. Interestingly, even a rate of 1 Gbps per agg box reduces the total completion time by more than 74% for the 1:4 oversubscribed scenario and 63% for the non-oversubscribed one (88% and 90% for the rate of 8 Gbps, respectively). This shows that, although the bandwidth per agg box is identical to the one used by the rack-level aggregator, performing aggregation at all network tiers (as opposed to just within a rack) reduces the pressure placed on the over-subscribed network core. Note that these results include *all* flows, not just the aggregation flows, which means that even flows that cannot be aggregated benefit from more efficient bandwidth usage.

In §4.2, we show that an agg box in our NETAGG prototype is able to aggregate data at a rate of 9.2 Gbps. Based on the results presented here, this is sufficient to obtain a significant reduction in flow completion time.

**Cost analysis.** Next we perform a simple cost analysis to understand the trade-off between deploying agg boxes versus increasing the network bandwidth. We consider three alternative options to NETAGG: a full-bisection network topology with 1 Gbps edge links (FullBisec-1G); a 1:4 over-subscribed network with 10 Gbps edge links (Oversub-10G); and a full-bisection network topology with 10 Gbps edge links (FullBisec-10G).

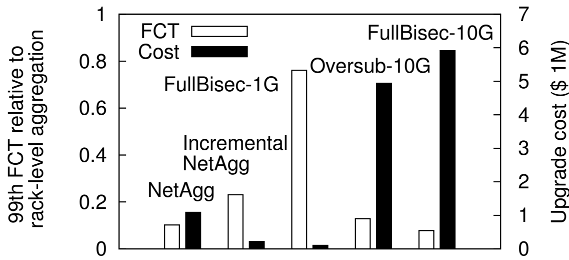


Figure 3: Performance and cost of different DC configurations

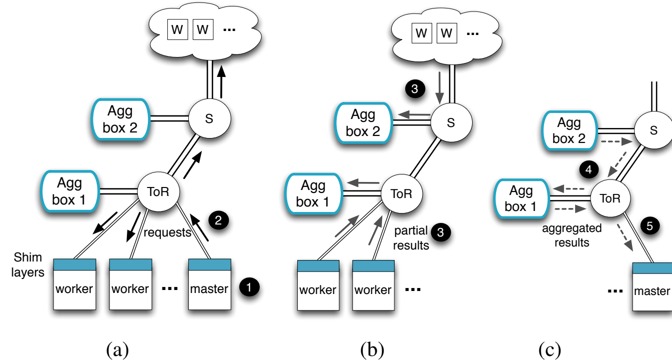


Figure 4: NETAGG middlebox deployment with sample workflow (numbers correspond to steps in workflow)

We compare the performance and cost when using rack-level aggregation in the three configurations above to deploying agg boxes using our approach in the base set-up (1:4 over-subscription and 1 Gbps network). We use the same workload as in the previous experiment, and we conservatively assume a configuration with a processing rate $R$=9.2 Gbps because this is similar to the rate achieved by our NETAGG prototype (§4.2). For NETAGG, we consider two deployment options: NetAgg, in which we assume that each switch is connected to an agg box, and Incremental-NetAgg, in which only the middle tier of switches (corresponding to 20% of all switches) is connected to agg boxes. We adopt the prices for network equipment from a recently published study [46]. We assume hardware specifications for servers and agg boxes as used in our evaluation testbed (see §4.2).

Figure 3 shows the performance improvement and upgrade costs with respect to the base set-up. As expected, upgrading to a 10 Gbps full-bisection network (FullBisec-10G) provides the largest benefit (92% reduction of FCT), but it also incurs the highest upgrade cost. FullBisec-1G has a much lower deployment cost but does not result in the same benefit (only 24% reduction).

In contrast, deploying NetAgg achieves almost the same performance improvement (88%) as FullBisec-10G and outperforms Oversub-10G (87%), with only a fraction of the cost (18% and 22%, respectively). Incremental-NetAgg is also a practical deployment option: it only incurs 4% of the cost of Oversub-10G but reduces FCT by 75%.

**Discussion.** While we compare the performance of on-path agg boxes against the performance of an upgraded network infrastructure, our solution remains complementary: even with more available network bandwidth, on-path aggregation can reduce bandwidth consumption and provide more bandwidth to other, non-partition/aggregation traffic flows.

In an upgraded network infrastructure (e.g. with 10 Gbps edge links), however, the performance of agg boxes should increase accordingly. As we describe in the next section, the aggregation on agg boxes is embarrassingly parallelisable, which means that agg boxes can exploit increasing numbers of CPU cores for higher performance. In addition, given the nature of aggregation, it is possible to *scale out* agg boxes by deploying multiple agg boxes connected to one network switch. We evaluate the performance implications of scaling out agg boxes in §4.

# 3. DESIGN AND IMPLEMENTATION

Next we describe the design of the NETAGG middlebox platform. We explain how it performs aggregation (§3.1), and give details on the implementation of aggregation boxes and shim layers (§3.2). We also report on two application case studies (§3.3).

## 3.1 Overview

As shown in Figure 4, a deployment of NETAGG in a DC consists of two main components, *agg boxes* and *shim layers*: (i) agg boxes are connected to a subset of the switches in the DC via high-bandwidth network links. They perform the on-path aggregation of application data according to aggregation functions; (ii) edge servers in the DC are fitted with shim layers that intercept request and result data flows, interacting with the agg boxes.

The agg boxes cooperate to establish an on-path *aggregation tree* that aggregates partial results from worker nodes before sending the final result to the master node. As discussed in §2, an aggregation tree permits NETAGG to exploit the larger core network bandwidth for data aggregation. Its construction relies on the associativity and commutativity of aggregation functions.

The workflow between agg boxes and shim layers for on-path aggregation follows the steps shown in Figs. 4a–4c:

1. A client submits a request to a master node, which partitions it into multiple sub-requests (Figure 4a).
2. The master node sends the sub-requests, which pass through the shim layer of the master node without modification, to a set of worker nodes.
3. The partial results generated by the workers are intercepted by the shim layers of the worker nodes. The shim layers redirect the data to the first agg box along the network path from the worker to the master node (Figure 4b). For example, agg box 2, connected to an aggregation switch, aggregates partial results from workers in other parts of the DC.
4. Each agg box aggregates the partial results according to the aggregation function (Figure 4c). It sends its partially aggregated results to the next agg box that is along the network path to the master node.
5. The agg box nearest to the master node (i.e. agg box 1 in the example) sends the fully aggregated results to the master node. The shim layer of the master node passes the results to the application.

**Multiple applications.** Multiple applications may be executing aggregation trees concurrently on agg boxes. Based on the knowledge of its child and parent nodes, an agg box therefore always forwards partial results to a chosen next-hop towards the master node along a given aggregation tree belonging to that application. The next agg box on-path is determined by hashing an application/request identifier. This ensures that partial data for a given application request traverses the same agg boxes.

**Multiple aggregation trees per application.** With a single per-application aggregation tree, it is not possible to exploit DC network topologies that support multiple routing paths and thus have higher core network bandwidth (see §2.3). NetAgg therefore supports multiple aggregation trees that are used concurrently for a given application, partitioning the aggregation load among the trees. Each aggregation tree uses a disjoint set of agg boxes (except for the agg box that is in the same rack as the master and the workers), exploiting different routing paths in a multi-path topology.

With multiple aggregation trees per application, the shim layers at the worker nodes partition partial results across the trees. Typically this can be achieved by hashing request identifiers (as in the case of online services) or keys in the data (as in the case of batch processing applications). When an application has multiple aggregation trees, the master node must perform a final aggregation step of the data returned by the roots of the trees.

**Multiple agg boxes per switch.** To increase the throughput of an agg box connected to a switch, it is possible to scale out processing by load-balancing aggregation computation across multiple agg boxes connected to the same switch. In this case, aggregation trees are assigned to agg boxes in a way that balances the load between them.

**Handling failures.** The NetAgg design uses a lightweight failure detection service, running at both the agg boxes and the master shim layer, that monitors the status of downstream agg boxes in the distributed aggregation tree. When a node $N$ (either an agg box or the master node) detects that a downstream agg box $F$ has failed, it contacts the child nodes (either agg boxes or the worker nodes) of $F$ and instructs them to redirect future partial results to $N$.

To avoid duplicate results, when $N$ contacts the downstream nodes of $F$, it also sends the last result that has been correctly processed, e.g. the last partial result received, so that already-processed results are not resent.

**Handling stragglers.** NetAgg is designed to be compatible with existing mechanisms used by applications to handle straggling worker nodes. For example, Hadoop uses speculative execution of backup tasks and reducers can start fetching data from completed mappers while waiting for stragglers or backup tasks to finish [25]. In this case, the agg box just aggregates available results, while the rest is sent directly to the reducer.

Since an aggregation tree involves multiple agg boxes, each agg box itself can potentially become a straggler, delaying the computation of the final result. To handle this scenario, NetAgg uses a similar mechanism to the one for failures: if a node detects that the downstream agg box $S$ is too slow (based on an application-specific threshold), it contacts the downstream nodes of $S$ to redirect future results. The difference with the failure protocol is that the redirection is only applied to results of the same request because the cause of straggling may be specific to it. However, if low performance is observed repeatedly across different requests, the agg box is considered permanently failed, and the failure recovery procedure described above is employed.

## 3.2 Implementation

We implement a prototype version of NetAgg in Java, which permits the agg boxes to execute unmodified aggregation functions of partition/aggregation applications written in Java. Agg boxes can host aggregation functions of multiple applications.

An important goal is for agg boxes to use available hardware resources efficiently in order to process data with high throughput. Their implementation is therefore data-parallel: they decompose aggregation functions and parallelise their execution across CPU cores using *cooperative scheduling*. In addition, we minimise the application-specific functions that agg boxes execute: they only receive and process data traffic and do not participate in control interactions of applications.

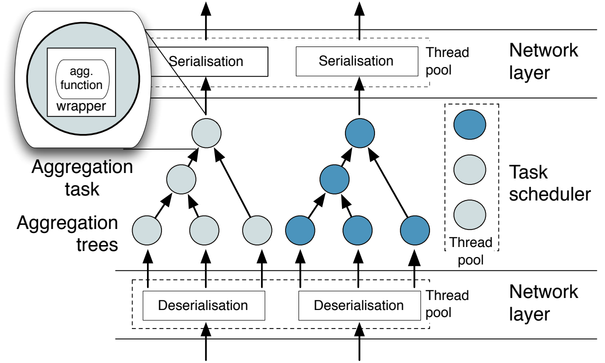


Figure 5: Architecture of a NetAgg agg box with two deployed applications (shading represents different aggregation functions)

For multiple applications to share a NetAgg deployment, it is necessary to schedule access to the limited CPU and bandwidth resources of agg boxes. Agg boxes implement prioritised access for certain classes of applications, such as latency-sensitive ones. They schedule the execution of aggregation computation belonging to different applications using *adaptive weighted fair scheduling*, taking different priorities into account.

To make it easy for applications to benefit from on-path aggregation, NetAgg transparently *intercepts* data flows and redirects them to agg boxes. Traffic is intercepted at the level of Java network sockets due to its well-defined interface across applications.

### 3.2.1 Agg boxes

We show the architecture of an agg box in Figure 5: (i) it executes *aggregation tasks* that wrap the applications' aggregation computations; (ii) the tasks are organised into a *local aggregation tree* that parallelises the aggregation function; (iii) tasks are scheduled cooperatively across CPU cores by a *task scheduler*; and (iv) a *network layer* serialises and deserialises data.

**Aggregation tasks.** An agg box represents computation as aggregation tasks, which are fine-grained compute units that can be scheduled in parallel on CPU cores.

Different applications require different interfaces for aggregation functions. An aggregation task therefore wraps aggregation computation using an *aggregation wrapper*. For example, a Hadoop aggregation wrapper exposes the standard interface of combiner functions, Combiner.reduce(Key k, List<Value> v), which enables agg boxes to run such functions without modification.

**Local aggregation trees.** Similar to how aggregation computation is decomposed across multiple agg boxes, the computation within a single agg box forms a *local* aggregation tree of tasks. Local aggregation trees have a large fan-in and are executed in a pipelined fashion by streaming data across the aggregation tasks. This allows for efficient and scalable aggregation, as aggregation executes in parallel and little data is buffered.

As shown in Figure 5, the intermediate tree nodes are aggregation tasks, acting as producers and consumer of data. Leaf nodes receive partial results from worker nodes or downstream agg boxes and send them to their parent aggregation tasks. Tasks are scheduled by the task scheduler (see below) when new input data is available and there is sufficient buffer space at the parent task. They ex-

ecute the aggregation function on the data from their child nodes. Partially aggregated results propagate up the tree until the root node obtains the final result for that tree. If the computation in the local aggregation tree is not fast enough to sustain the rate at which data arrives from the network, a back-pressure mechanism ensures that the workers reduce the rate at which they produce partial results.

**Task scheduler.** To reduce the overhead of thread synchronisation, aggregation tasks are executed by a task scheduler using *cooperative scheduling*: when aggregation is possible, a task is submitted to a task queue and waits to be scheduled. The scheduler assigns tasks to threads from a fixed-sized thread pool and runs the tasks to completion. We assume that aggregation functions are well-behaved and terminate—we leave mechanisms for isolating faulty or malicious aggregation tasks to future work.

If there are multiple applications using an agg box, resources must be shared to achieve acceptable processing times for aggregation computation. For this purpose, an agg box maintains a separate task queue for each application and adopts a *weighted fair queuing* policy over these queues. This enforces weighted fair shares among applications, similar to other cluster schedulers [60]. When a thread becomes available, the scheduler offers that thread to a task of application $i$ with probability $w_i / \sum_{i=1}^{n} w_i$ that is proportional to its target allocation, where $w_i$ is application $i$'s weight and $n$ is the number of applications on the agg box.

Resource sharing must take the heterogeneity of aggregation tasks into account: the computation and bandwidth requirements of tasks vary depending on the application. To handle this, the scheduler periodically adapts the weights of application $i$ according to the task execution time $\bar{t}_i$ measured at runtime. The intuition is that if an application usually spends twice the time finishing tasks compared to another, the scheduler needs to halve the weight of that application to achieve the targeted proportional share. More formally, given a target resource share $s_i$ for application $i$, $w_i$ is set to $\frac{s_i}{\bar{t}_i} / \sum_{i=1}^{n} \frac{s_i}{\bar{t}_i}$. Our implementation uses a moving average to represent the measured task execution time—the experiments in §4 show that this is sufficient in practice.

**Network layer.** Instead of relying on a potentially wasteful application-specific network protocol, such as HTTP or XML, agg boxes transfer data with an efficient binary network protocol using KryoNet [39], an NIO-based network communication and serialisation library for Java. The shim layers also maintain persistent TCP connections to the agg box and parallelise deserialisation using a thread pool.

Since the network data must be deserialised before the aggregation wrapper can call the aggregation function, the network layer of the agg box includes a serialiser/deserialiser taken from the application. For example, to support the aggregation of Hadoop key/value pairs, the agg box uses Hadoop's `SequenceFile` reader and writer classes for serialisation/deserialisation.

### 3.2.2 Shim layers

The shim layers control the redirection of application data and manage the collection of partial results.

**Network interception.** A shim layer intercepts network traffic at the level of network sockets by wrapping the actual Java network socket class in a NETAGG socket. By using Java's ability to change the default socket implementation via the `SocketImplFactory`, applications transparently generate an instance of the custom NETAGG socket class when a new socket is created.

**Partial result collection.** A challenge is that agg boxes must know when all partial results were received before executing the aggregation function but, for many applications including Apache Solr, it is not known ahead of time how many partial results will be returned by the worker nodes. We solve this problem by having the shim layer of the master node maintain state about ongoing requests. After intercepting an incoming request, the shim layer records information about the request, typically found in headers, such as the number of partial results, and sends this information to agg boxes.

| *App.-specific* NETAGG *code* | *Solr* | *Hadoop* |
|---|---|---|
| Agg box (serialisation) | 8 | 93 |
| Agg box (aggregation wrapper) | 156 | 142 |
| Shim layer | 449 | 528 |
| *Total* | 613 | 763 |
| *Relative to* NETAGG *code base* | 13% | 16% |
| *Relative to application code base* | 0.19% | 0.03% |

Table 1: Lines of application-specific code in NETAGG

**Empty partial results.** The master node expects to receive partial results from all worker nodes, but with NETAGG this is no longer the case: since the partial results have already been aggregated, only a single fully aggregated result is returned. This means that the shim layer at the master node must emulate empty results from all but one worker, which will include the fully aggregated results. The aggregation logic in the master node is not be affected by the empty results because we assume that aggregation functions are commutative and associative.

## 3.3 Application deployments

To demonstrate the generality of NETAGG, we describe its deployment with two partition/aggregation applications: Apache Solr, a distributed full-text search engine [11], and Apache Hadoop [10], a data-parallel map/reduce processing framework. NETAGG can support both applications after providing application-specific serialiser/deserialiser, aggregation wrapper and shim layer code. The required implementation effort is summarised in Table 1.

**Apache Solr** performs full-text search across multiple *backend* server nodes, acting as workers. Clients send small requests to a *frontend* node, i.e. the master node, which dispatches sub-requests to the backend nodes. The backends return partial search results to the frontend node, which combines them into a ranked result using an aggregation function.

To support Solr, NETAGG requires around 600 lines of code. The aggregation wrapper wraps the custom `QueryComponent` class, which allows user-defined aggregation of the result data. The deserialiser buffers all partial results before invoking the local aggregation tree: this is feasible because results are only of the order of hundreds of kilobytes.

**Apache Hadoop** uses worker nodes to execute mappers, which produce partial results for partitions of the input data. Partial results are aggregated by reducers.

There are fewer than 800 lines of code needed to support Hadoop. The aggregation wrapper implements the Hadoop interface for combiner functions. The deserialiser for Hadoop is slightly more complex than for Solr: as chunks of key/value pairs are processed by agg boxes in a streaming fashion, the deserialiser must account for incomplete pairs at the end of each received chunk.

## 4. EVALUATION

We begin our evaluation by extending the simulation study from §2.4. In §4.2, we show the experimental results obtained by deploy-

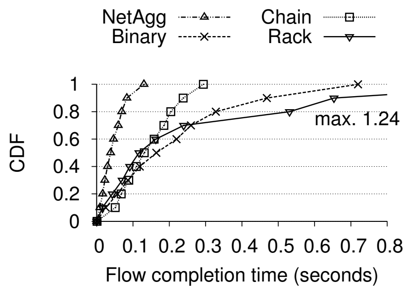


Figure 6: CDF of flow completion time of all traffic

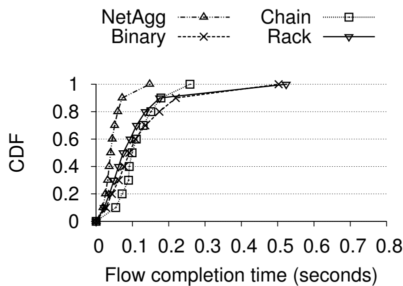


Figure 7: CDF of flow completion time of non-aggregatable traffic

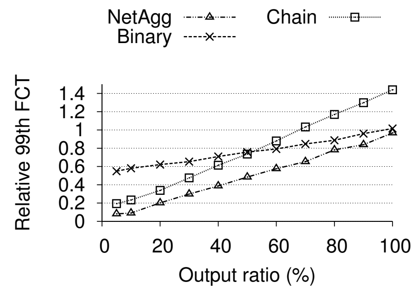


Figure 8: Flow completion time relative to baseline with varying output ratio $\alpha$

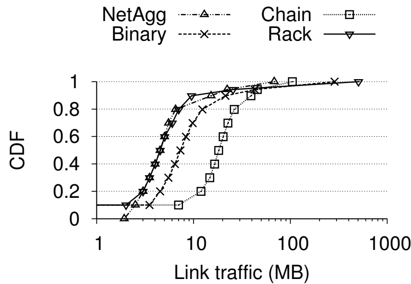


Figure 9: CDF of link traffic ($\alpha = 10\%$)

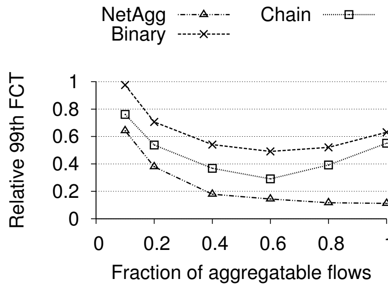


Figure 10: Flow completion time relative to baseline with varying fraction of aggregatable traffic

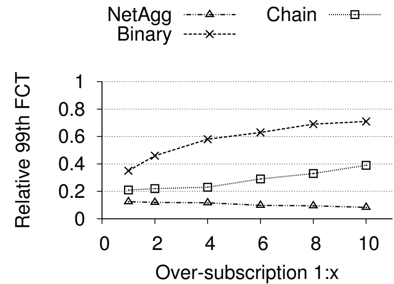


Figure 11: Flow completion time relative to baseline with different over-subscription ($\alpha = 10\%$)

ing our NETAGG middlebox prototype with different applications on a 34-server testbed.

## 4.1 Simulation results

We simulate a three-tier, multi-rooted network topology based on recently proposed scalable DC architectures [2, 29]. The network consists of 320 20-port switches that connect 1,024 servers using 1 Gbps links, unless noted otherwise. As a default, we assume a 1:4 over-subscription ratio at the ToR tier, varying it from 1:1 to 1:10 in one of the experiments. We use a packet-level simulator (OMNeT++), which implements TCP max-min flow fairness and uses standard Equal Cost Multi Path (ECMP) for routing.

Unless noted otherwise, we assume that each agg box is connected to a switch through a 10 Gbps link, and we assume that it can process network flows at 9.2 Gbps. As we show in §4.2, this is representative of the performance of our current NETAGG prototype implementation.

We use a synthetic traffic workload, modelled after published network traces from a cluster running large data mining jobs [29]. The sizes of flows follow a Pareto distribution with a mean of 100 KB and a shape parameter of 1.05. The number of flows is chosen so that the load at the edge links is 15% [5]. Except when simulating straggler nodes, all flows start at the same time, which is a worst case for network contention. We also ran experiments using dynamic workloads with various arrival patterns, obtaining comparable results (between 1%–4% of the reported FCT values).

A typical partition/aggregation application generates a mix of flows, of which only a fraction can be aggregated. For example, some of the network flows in Hadoop contain HDFS data, which cannot be aggregated. To account for this, we generate a mixed workload in which only 46% of flows are aggregatable, while the rest constitutes non-aggregatable background traffic. This split is consistent with evidence reported by Facebook [19].

The number of workers generating flows follows a power-law distribution where 61% of requests or jobs have fewer than 10 workers, consistent with a recent study on Microsoft and Facebook production clusters [7]. Workers are deployed using a locality-aware allocation algorithm that greedily assigns workers to servers as close to each other as possible. Unless stated otherwise, we use an aggregation output ratio $\alpha$, defined as the ratio between the output and the input data sizes, of 0.1 (in line with the workloads presented in §2.1).

**Baselines.** We use the *flow completion time* (FCT) as our evaluation metric because our goal is to improve network performance. Since our focus is on network bottlenecks that increase FCTs for individual flows, we report the 99$^{th}$ percentile of FCT, unless stated otherwise. We compare the performance of NETAGG against the aggregation strategies described in §2.2: rack-level aggregation (rack), a binary aggregation tree ($d = 2$; binary) and chain aggregation ($d = 1$; chain). Unless reporting absolute FCTs, we normalise the performance of the other strategies against rack.

**Distribution of flow completion times.** We report the CDF of the absolute FCTs for *all* flows in Figure 6. As explained in §2.2, binary and chain increase link utilisation, which in turn reduces the bandwidth available for other flows. This explains why they reduce the FCT tail but worsen the median. In contrast, NETAGG improves the performance for all flows by reducing the traffic of the aggregatable flows.

Interestingly, NETAGG also improves the performance of non-aggregatable flows, as shown in Figure 7. The reason is that by reducing the traffic of aggregatable flows, more bandwidth becomes available for other flows.

**Aggregation output ratio.** In Figure 8, we show the impact of the output ratio $\alpha$. We vary $\alpha$ from 0.05 (i.e. high data reduction that emulates $n$-to-1 aggregation patterns such as top-k, max or count)

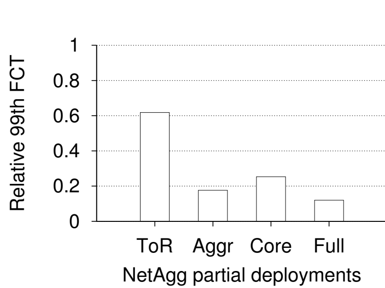

Figure 12: Flow completion time relative to baseline with different partial NETAGG deployments

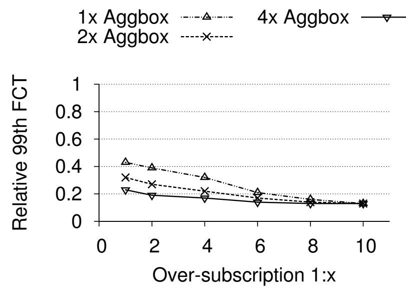

Figure 13: Flow completion time relative to baseline in 10 Gbps network with varying over-subscription

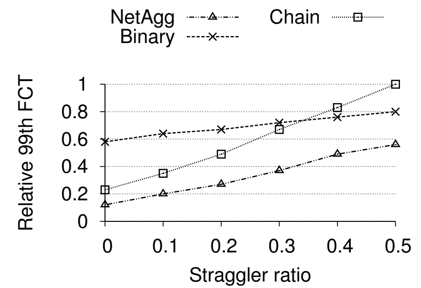

Figure 14: Flow completion time relative to baseline with varying stragglers

to 1 (i.e. no aggregation is possible). As expected, the benefits of NETAGG decrease for high values of $\alpha$. When $\alpha$ is small, NETAGG significantly reduces the FCT: for $\alpha = 10\%$, NETAGG reduces the 99$^{th}$ FCT compared to rack by 88% (by 80% and 52% for binary and chain, respectively).

For very high values of $\alpha$, the performance of NETAGG is only marginally better than rack. The reason is that, when there is little or no aggregation, the use of NETAGG penalises throughput because it forces the cross-rack traffic still to be processed by the agg box (at a maximum rate of 9.2 Gbps in our experiments), which constraints the overall throughput, without providing a benefit in terms of data reduction. However, since production traces show that most applications exhibit an output ratio lower than 10% [12, 18], a small $\alpha$ is representative of typical workloads, and we adopt $\alpha = 10\%$ in the rest of the experiments.

It is also worthwhile to analyse why chain is always outperformed by rack for $\alpha > 70\%$. The reason of this counter-intuitive behaviour is that, as explained in §2.2, chain utilises more link bandwidth compared to rack. As shown in Figure 9, with $\alpha = 10\%$, the median link traffic for chain is 4× higher than for rack (2.5× for binary, respectively). When the network is highly loaded and $\alpha$ is large, this aspect dominates, thus lowering throughput.

**Fraction of aggregatable flows.** As described in §2.4, a typical DC traffic workload contains a mix of flows, of which only a fraction can be aggregated. We conduct a sensitivity analysis in which we vary the fraction of aggregatable flows from 0.1 to 1 (i.e. all flows can be aggregated).

Figure 10 shows the 99$^{th}$ percentile of FCT relative to rack. With more aggregatable flows, the benefit of all three aggregation strategies increases. With more than 60% of aggregatable flows, however, the effectiveness of binary and chain starts to decrease again because their more wasteful usage of network bandwidth introduces network bottlenecks. In contrast, NETAGG maintains the lowest FCTs, all the way to a fully-aggregatable workload.

**Over-subscription.** We quantify the performance impact of the bisection bandwidth on NETAGG and our baselines in Figure 11. We vary over-subscription from 1:1 (i.e. full-bisection bandwidth) to 1:10. As expected, NETAGG performs best when over-subscription is high. By aggregating flows along the network paths, it reduces congestion in the core, alleviating network bottlenecks.

However, the use of NETAGG is beneficial even for networks with full-bisection bandwidth. In a full-bisection network, the inbound bandwidth of the rack and the master becomes the bottleneck. By aggregating the traffic along the network path, NETAGG reduces the congestion at the master, thus decreasing the completion time.

**Partial deployment.** One important question is whether it would be possible to achieve a performance close to NETAGG by simply deploying a 10 Gbps server in each rack and use it for rack-level aggregation. More generally, we want to understand what should be the best deployment configuration if we had only a limited number of agg boxes.

In Figure 12, we compare the performance of a full NETAGG deployment against three configurations: (i) agg boxes only at the ToR switches; (ii) only at the aggregation switches; and (iii) only at the core switches. The results show that the biggest improvement is achieved with agg boxes at the aggregation or core network tiers—deploying them at the rack tier provides limited benefits, improving the FCT by 38% against 83% and 75%, respectively.

This comparison, however, does not take into account that in a DC topology the number of switches at each tier varies significantly. For example, in our simulated topology, we have 128 ToR switches, 128 aggregation and 64 core switches. Therefore, we fix the number of agg boxes to 64, and we measure the performance when deploying them (i) at the core tier only; (ii) uniformly distributed at the aggregation tier; and (iii) uniformly distributed at the two tiers.

Interestingly, the first configuration achieves the largest improvements (75%) while the other two achieve 29% and 43%, respectively. The reason is that the core tier has the best opportunity for aggregating data because it intercepts more flows originating from the workers. This result is important because it shows the benefit of aggregating in the network and not just at the edge. Furthermore it also demonstrates that NETAGG can be deployed incrementally while still yielding most of the benefits.

**Upgrading to a 10 Gbps network.** In our next simulation experiment, we explore the performance of NETAGG when deployed in a 10 Gbps network. We use the same topology and workload as before but with 10 Gbps links connecting servers to ToR switches.

In Figure 13, we vary the over-subscription from 1:1 to 1:10. For large over-subscription values, NETAGG provides significant benefits compared to rack. Although edge servers have the same bandwidth as agg boxes, the over-subscription creates high contention in the core, which limits the throughput of rack. This shows the benefits of aggregating flows at each network hop as opposed to the edge servers only.

For smaller over-subscription values, the agg box processing rate becomes the bottleneck, reducing benefit. To further increase performance, we explore a *scale out* configuration in which we attach 2 or 4 agg boxes to each switch. We use a strategy similar to ECMP to assign flows of the same application to the same set of agg boxes. As the results show, two agg boxes are sufficient to reduce the FCTs by up to 85% compared to rack. This demonstrates

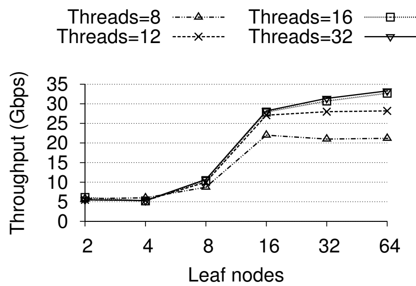


Figure 15: Processing rate of an in-memory local aggregation tree on a 16-core server

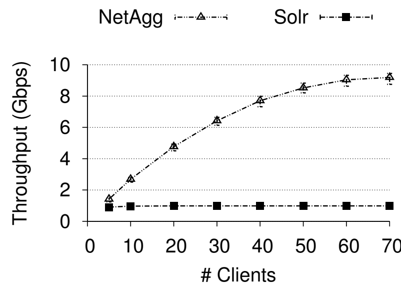


Figure 16: Network throughput against number of clients (Solr)

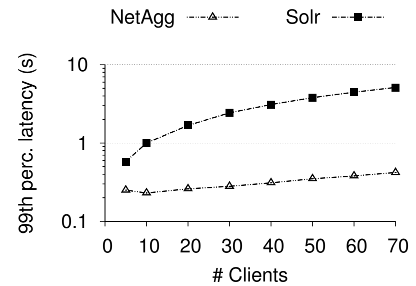


Figure 17: Response latency against number of clients (Solr)

the value of scaling out, making NETAGG compatible with future network upgrades.

**Stragglers.** To quantify the impact of straggling workers on our solution, we run an experiment in which we artificially delay the starting time of some of the flows of a given request or job, following the distribution reported in the literature [8]. The results in Figure 14 show that for reasonable ratios of stragglers, NETAGG can still provide significant traffic reduction. As expected, however, with a higher number of stragglers, the benefits of NETAGG decrease as there are fewer opportunities to aggregate data.

## 4.2 Testbed results

Next we describe the results of our testbed experiments designed to evaluate the effectiveness of our NETAGG middlebox prototype. To demonstrate the flexibility of NETAGG, we experiment with an online application, the Apache Solr distributed text search engine [11], and a batch-processing application, the Apache Hadoop map/reduce framework [10]. The two applications set different objectives for NETAGG, namely increased request throughput (Solr) and reduced job completion time (Hadoop).

We first evaluate the applications in isolation (§4.2.1 and §4.2.2) and then, to demonstrate NETAGG's ability to host multiple applications, run them simultaneously (§4.2.3).

**Testbed set-up.** The experiments are performed using a prototype NETAGG implementation deployed on a 34-server testbed across two racks. Each rack contains one 16-core 2.9 Ghz Intel Xeon server with 32 GB of RAM, acting as a master node, and ten 4-core 3.3 Ghz Intel Xeon servers with 8 GB of RAM, acting as the workers. In addition, each rack has five 4-core 1.8 Ghz AMD Opteron servers, generating the client queries in the Solr experiments. All servers have 1 Gbps network links to their ToR switches. Agg boxes have the same hardware as the master nodes and are attached to the ToR switches via 10 Gbps links.

**Performance of local aggregation tree.** We first conduct a micro-benchmark that evaluates the processing performance of differently-sized in-memory local aggregation trees. The experiments use a 16-core 2.9 Ghz Intel Xeon server with 32 GB of RAM. For an $n$-input binary tree, $n$ worker threads feed deserialised data to the tree. We use the Hadoop WordCount workload (see below) with an output ratio of $\alpha = 10\%$.

Figure 15 shows the processing throughput for different thread pool sizes when varying the number of leaves in a tree ($L$). We only consider binary trees and, hence, the number of internal nodes, i.e. the aggregation tasks, is equal to $L-1$. We observe that 8 leaves are sufficient to saturate a 10 Gbps link. When the number of leaves increases, more aggregation tasks can be executed concurrently, and the throughput increases accordingly.

### 4.2.1 Apache Solr

We begin our testbed evaluation with Apache Solr in a single-rack scenario with 1 frontend, 10 backend and 5 client servers. We load the Wikipedia dataset, a snapshot of all Wikipedia pages in XML format from June 2012 [57], into the backends, and each client continuously submits a query for three random words. To generate different workloads, we vary the number of concurrent clients generating requests. Unless stated differently, error bars indicate the 5$^{th}$ and 95$^{th}$ percentiles.

We use two aggregation functions with different computational costs, representing extremes for functions commonly used in search engines. The first function, `sample`, has low computational cost: it returns a randomly chosen subset of the documents to the user according to a specified output ratio $\alpha$, which therefore controls the amount of data reduction: a lower value of $\alpha$ means more aggregation, i.e. less data is returned to the clients.

The second function, `categorise`, is CPU-intensive: it classifies Wikipedia documents according to their *base categories* [56] and returns the top-k results per category. Each base category contains several sub-categories, and categorisation is performed by parsing the document content for category strings and determining the base category for the majority of strings.

**Throughput.** Figure 16 shows the median throughput with an increasing number of clients for Solr deployed on NETAGG. To test the capability of NETAGG to process at line rate, we use the `sample` function for aggregation with a fixed output ratio of $\alpha = 5\%$ to prevent the link to the frontend from becoming a bottleneck. For comparison, we also show the throughput of plain Solr.

For plain Solr, the throughput with 5 clients is limited by the client workload. As the number of clients increases, the throughput also increases until saturation. For 10 clients, packets from the backends start to queue on the 1 Gbps link to the frontend until the system completely saturates for 30 clients (processing at a maximum rate of 987 Mbps). After that, adding clients results in queuing at the frontend without improving the throughput.

With NETAGG, the throughput grows steadily up to 50 clients and then starts to saturate, reaching a median throughput of 9.2 Gbps for 70 clients. This corresponds to a 9.3× increase compared to Solr. After this, the throughput is bottlenecked by the incoming link bandwidth of the agg box.

**Latency.** Besides improving the throughput, NETAGG also reduces the request latency. Figure 17 shows the 99$^{th}$ percentile of the response times when varying the number of clients. For Solr, as the throughput of the frontend is limited to 1 Gbps, response times rise significantly when the workload increases. NETAGG can serve a higher load with low response times: with 70 clients, the response time for Solr is 5.1 s, while it only increases to 0.4 s for NETAGG.

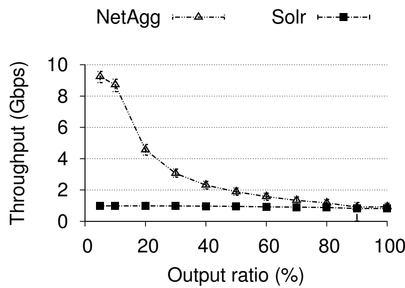


Figure 18: Network throughput against outgoing ratio (Solr)

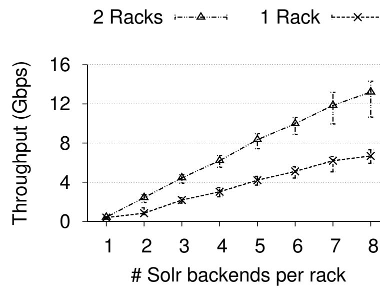


Figure 19: Throughput against number of backend servers per rack (Solr)

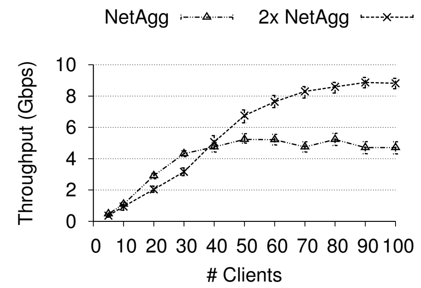


Figure 20: Agg box scale out for CPU-intensive aggregation (Solr)

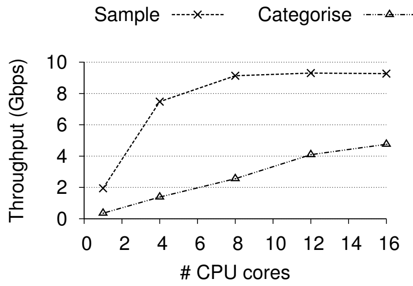


Figure 21: Throughput against number of CPU cores (Solr)

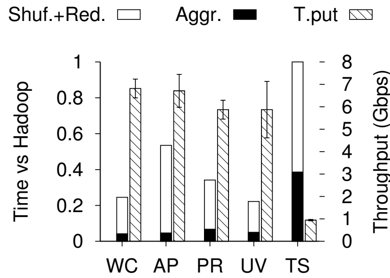


Figure 22: Performance of Hadoop benchmarks

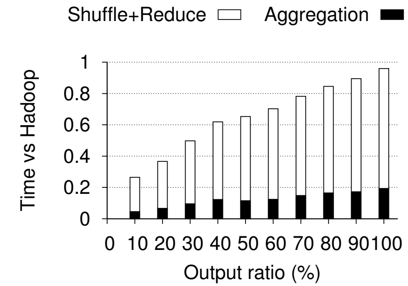


Figure 23: Shuffle and reduce time against output ratio (Hadoop)

This shows that NETAGG achieves low response times because it removes congestion from the network link to the frontend.

**Output ratio.** Next we vary the output ratio $\alpha$ of Solr requests with the sample aggregation for a fixed client population of 70. Figure 18 shows the throughput for Solr and NETAGG. Since the performance of Solr is network-bound, the throughput does not depend on the output ratio. For NETAGG, a higher output ratio also increases the utilisation of the network path between the agg box and the frontend, which remains limited to 1 Gbps. Therefore the effectiveness of NETAGG decreases. As described in §2.1, output ratios in production environments typically range from 5% to at most 40%, in which case NETAGG can offer significant benefit.

**Two-rack deployment.** To investigate the performance of NETAGG with multiple agg boxes, we extend our set-up to two racks, each with one agg box. We also add a second Solr deployment to generate a higher request load. We vary the number of backends per Solr deployment, which are divided equally between the racks to generate cross-rack traffic.

Figure 19 compares the throughput of a single agg box in one rack to the aggregate throughput of two agg boxes in two racks. For both configurations, throughput scales linearly with the number of backends. With two racks, the agg boxes serve twice as many backends compared to one rack, and hence their aggregate throughput doubles. This shows the ability of NETAGG to operate in larger deployments.

**Scale out.** To show the feasibility of increasing the performance of NETAGG by connecting multiple agg boxes to a single switch, we conduct an experiment in which we compare the performance of a single agg box with two agg boxes attached to the same switch. Requests are split equally between the agg boxes by hashing request identifiers. We use the computationally-expensive categorise aggregation function to ensure that the agg box is the bottleneck.

The results in Figure 20 confirm our expectation: they show that, by adding a second agg box, the throughput increases until the agg boxes become network-bound.

**Scale up.** To demonstrate the effectiveness of data-parallel processing on agg boxes, we increase the number of active CPU cores on a single agg box from 1 to 16 for both of our aggregation functions. The results in Figure 21 show that, while the performance of the computationally-inexpensive sample function is network bound, the performance for the categorise function increases linearly with more CPU cores due to increased parallelism.

### *4.2.2 Apache Hadoop*

Next we investigate the performance of NETAGG when used with a batch processing application (Apache Hadoop). We deploy Hadoop in one rack with 8 mappers and 1 reducer, with a single aggregation tree. The workload consists of a set of benchmarks: (i) WordCount (WC), counting unique words in text; (ii) AdPredictor (AP), a machine learning job for generating click-through predictions from search engine web logs [28,38]; (iii) PageRank (PR), an implementation of the PageRank algorithm [33]; (iv) UserVisits (UV), a job for computing ad revenue per source IP address from web logs [12]; and (v) TeraSort (TS), a sorting benchmark with an identity reduce function [33].

**Job completion time.** We deploy each benchmark job on plain Hadoop and on Hadoop with NETAGG. For each job, we measure (i) the total shuffle and reduce time (SRT) relative to Hadoop, which includes the time spent at the agg box (AGG) and at the reducer; and (ii) the corresponding processing throughput at the agg box. We ignore the map phase because it is not affected by NETAGG.

Figure 22 shows the shuffle and reduce times (normalised with respect to plain Hadoop) and the agg box processing rate. NETAGG reduces SRT up to 4.5× compared to plain Hadoop. Only

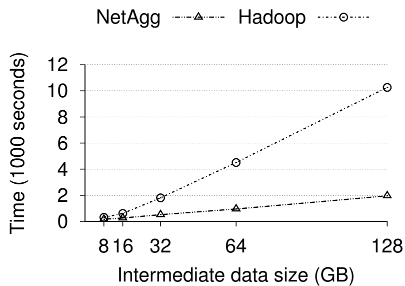


Figure 24: Shuffle and reduce time against intermediate data sizes (Hadoop)

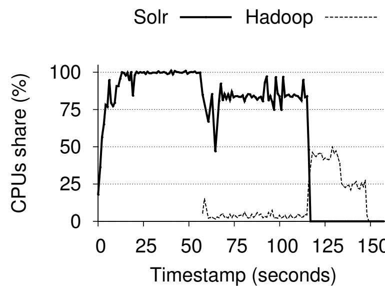


Figure 25: CPU resource fair sharing on NETAGG (Solr and Hadoop) achieved by a no-adaptive scheduler.

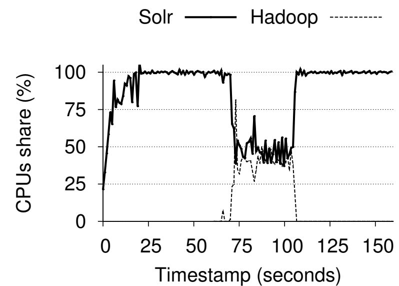


Figure 26: CPU resource fair sharing on NETAGG (Solr and Hadoop) achieved by an adaptive scheduler

for TS, there is no benefit because sorting does not reduce data. AP exhibits a speed-up of only 1.9× because the benchmark is compute-intensive.

Notably, in all cases, the time spent at the agg box (AGG) is a small fraction of the total SRT. This is due to the fact that the reducer is unaware that the results received from the agg box are already final and, regardless, reads them again. This is a conscious design decision because it makes agg boxes transparent to applications—with more invasive modifications of the application, the end-to-end SRT can be reduced to be closer to the AGG time.

The figure also shows that the agg box processes traffic at around 6 Gbps for almost all jobs—the processing rate for TS is lower due to a bottleneck at the reducer.

**Output ratio.** Next we compare the performance of Hadoop and Hadoop deployed with NETAGG for different output ratios $\alpha$, obtained by varying the repetition of words in the input for the WC job. The size of the input data is 8 GB, and we measure SRT.

Figure 23 shows that relative SRT increases with higher output ratios. This is due to the fact that, with higher ratios, there are more results that are written to disk. A decrease in the output ratio, however, does not alleviate the network bottleneck at the reducer, which has to receive data from all the mappers over a 1 Gbps link.

For all output ratios, NETAGG improves performance over plain Hadoop, up to a factor of 3.7× for $\alpha$=10%. The improvement is highest for low output ratios for the same reason as in the case of Solr: performance eventually becomes restricted by the limited edge bandwidth.

**Data size.** We explore how the performance benefit of NETAGG is affected by the amount of data processed by the reducer. For this experiment, we fix the output ratio at 10%. Figure 24 shows the absolute shuffle and reduce times for different intermediate data sizes, ranging from 8 GB to 128 GB.

As we increase the amount of intermediate data, the effect of the shuffle phase on the overall job completion time becomes more pronounced because more data is sent across the network. Hence, the benefit of NETAGG increases with more intermediate data, up to a factor of 5.2× for 128 GB. In addition, the reducer receives only a small fraction of the already reduced intermediate data. This reduces processing time as well as CPU cycles and disk I/O.

### 4.2.3 *Multiple applications*

In our final experiment, we evaluate the behaviour of NETAGG in the presence of multiple deployed applications, focusing on the fairness that the adaptive task scheduler of an agg box can achieve (§3.2.1). For this, we execute a Hadoop job on NETAGG while running a Solr deployment at the same time. We route all aggregation traffic through a single agg box and measure the CPU share of each application. As the resource consumption of a single Solr task is significantly higher than that of a Hadoop one, the task scheduler must be able to account for this heterogeneity.

Figure 25 shows the CPU usage of the two applications when the task scheduler uses regular weighted fair queuing with fixed weights, which are set according to application priorities (see §3.2.1). Although each application is assigned a desired utilisation of 50%, this is not reflected in the achieved CPU usage: a Solr task takes, on average, 34 ms to run on the CPU, while a Hadoop task runs only for 2 ms. Fixed weights therefore lead to a starvation of the Hadoop tasks, which exhibit low performance.

In contrast, Figure 26 shows the same set-up using our adaptive task scheduler, which adjusts weights based on the actual CPU consumption by applications. As a result, CPU usage is split fairly between the processing of aggregation tasks for Solr and Hadoop.

## 5. RELATED WORK

**Middleboxes.** Researchers have proposed efficient software middlebox platforms to process network flows using commodity hardware [50, 52]. ClickOS [43], for example, improves the network performance of Xen using netmap [49]. In contrast to NETAGG, these platforms operate at the packet level, which makes them unsuitable to aggregate payload data in application flows.

A flow-based middlebox platform is provided by xOMB [9], which can process application protocols such as HTTP. Yet its focus remains on a small set of network-level services, whereas NETAGG can host a number of application-specific aggregation functions on middleboxes to improve application performance. FlowOS [14] runs in kernel space and provides zero-copy flow construction for multiple, independent flows. However, FlowOS does not support aggregation across multiple, dependent flows, as in NETAGG.

Recently, SDN-based techniques were used to manage the traffic to and from middleboxes [48]. As part of NETAGG, we could use a similar approach to relay traffic to agg boxes instead of employing shim layers. However, shim layers selectively redirect traffic, not only based on packet headers, but also depending on application data semantics, which requires deserialising the application data.

**Data aggregation.** In map/reduce [25], aggregation of single map tasks through combiner functions is a common technique to help reduce network load and job execution time. To reduce the traffic at the network core, additional aggregation steps at one of the rack servers can be performed [40, 44, 59]. At scale, however, rack-level aggregation provides only limited opportunities for data reduction and is often bottlenecked by the lower edge bandwidth available at the servers (see §2).

Tyson et al. [21] describe Hadoop Online, an extension to map/reduce that allows reducers to start reducing intermediate results as soon as they become available. While this is orthogonal to the goals of NETAGG, our agg boxes use a similar technique to aggregate partial results with a streaming aggregation tree.

Camdoop [22] proposes an extreme solution to improve the performance of map/reduce by adopting a direct-connect DC topology. All traffic is forwarded between servers without switches, which can therefore aggregate the traffic. This approach, however, requires a custom network topology and redeveloped DC applications. In contrast, NETAGG is compatible with today's DCs and supports existing applications.

Aggregation was successfully applied in other domains, including wireless sensor networks and Internet systems. In sensor networks, packet-based aggregation is used to improve the efficiency of data-collection protocols, primarily targeting energy savings [34, 41]. Internet-scale aggregation systems use overlay networks to collect data in a scalable fashion [15, 35, 58]. Neither use of aggregation requires high throughput, a key design requirement of NETAGG. For example, the design proposed by Yalagandula and Dahlin [58] uses a DHT overlay in a WAN, which is a structure optimised for scale and churn, not throughput.

**Multi-point flow scheduling.** Recent work considered how to schedule multi-point flows efficiently in DCs [19, 20]. While we show an approach that improves network performance when the flows can be aggregated (*many-to-one*), we also note that application-specific middleboxes can implement efficient versions of multicast or broadcast protocols (*one-to-many*). This would enable further performance improvement of iterative applications with a distributed broadcast phase, such as graph processing or logistic regression [20].

**Traffic compression.** There are a variety of techniques that have been proposed to compress in-network traffic, mainly for the purposes of reducing bandwidth usage: network coding [1], traffic redundancy elimination [6], and even multicast [26, 55] are prime examples. These techniques are therefore complementary to our work, but do not address NETAGG's goal of supporting application-specific aggregation functions.

# 6. CONCLUSIONS

Many applications in today's DCs operate in a partition/aggregation fashion. We observe that in typical workloads, the aggregation functions are associative and commutative, and exhibit high data reduction. This motivates us to explore a novel point in the network design space in which data are aggregated on the network path before reaching the end hosts.

We describe the design and implementation of NETAGG, an on-path aggregation service that transparently intercepts aggregation flows at edge servers using shim layers and redirects them to aggregation nodes (agg boxes). NETAGG is designed to aggregate data of multiple applications at high rate by decomposing aggregation computation within and across agg boxes to exploit parallelism. We evaluate the effectiveness of NETAGG using a combined approach of simulations and a prototype implementation deployed with real-world applications. Our results show that NETAGG outperforms typical DC set-ups while incurring little extra deployment cost.

This work is part of a larger effort aimed at bridging the gap between networking and applications by enabling application-specific in-network processing [23]. Our ultimate goal is to provide novel expressive high-level abstractions coupled with efficient low-level mechanisms that allow users to deploy application-specific code within the network in a safe and efficient way.


# Acknowledgements

The authors wish to thank the anonymous reviewers and our shepherd, Christian Kreibich, who provided valuable feedback and advice. This work was supported by grant EP/K032968 ("NaaS: Network-as-a-Service in the Cloud") from the UK Engineering and Physical Sciences Research Council (EPSRC). Luo Mai is supported by a 2012 Google European Doctoral Fellowship in Cloud Computing.


# 7. REFERENCES


[1] R. Ahlswede, N. Cai, S.-Y. Li, and R. W. Yeung. Network Information Flow. *IEEE Transactions on Information Theory*, 46(4), 2000.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *NSDI*, 2012.

[6] A. Anand, V. Sekar, and A. Akella. SmartRE: An Architecture for Coordinated Network-Wide Redundancy Elimination. In *SIGCOMM*, 2009.

[7] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[8] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *OSDI*, 2010.

[9] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *ANCS*, 2012.

[10] Apache Hadoop. `http://hadoop.apache.org`.

[11] Apache Solr. `http://lucene.apache.org/solr`.

[12] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs Scale-out for Hadoop: Time to rethink? In *SOCC*, 2013.

[13] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[14] M. Bezahaf, A. Alim, and L. Mathy. FlowOS: A Flow-based Platform for Middleboxes. In *HotMiddlebox*, 2013.

[15] J. Cappos and J. H. Hartman. San Fermın: Aggregating Large Data Sets Using a Binomial Swap Forest. In *NSDI*, 2008.

[16] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management. In *SIGMOD*, 2013.

[17] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB*, 2008.

[18] Y. Chen, A. Ganapathi, R.Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS*, 2011.

[19] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *SIGCOMM*, 2013.

[20] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *SIGCOMM*, 2011.

[21] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *NSDI*, 2010.

[22] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *NSDI*, 2012.

[23] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NaaS: Network-as-a-Service in the Cloud. In *Hot-ICE*, 2012.

[24] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *NIPS*, 2012.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[26] C. Diot, W. Dabbous, and J. Crowcroft. Multipoint Communication: A Survey of Protocols, Functions, and Mechanisms. *J-SAC*, 15(3), 1997.

[27] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *CCR*, 36(1), 2006.

[28] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale Bayesian Click-through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In *ICML*, 2010.

[29] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[30] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, 2009.

[31] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *SIGCOMM*, 2011.

[32] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm, 2000. IETF RFC 2992.

[33] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *ICDE Workshops*, 2010.

[34] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed Diffusion for Wireless Sensor Networking. *ToN*, 11(1), 2003.

[35] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. STAR: Self-tuning Aggregation for Scalable Monitoring. In *VLDB*, 2007.

[36] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up Distributed Request-Response Workflows. In *SIGCOMM*, 2013.

[37] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *SIGCOMM*, 2008.

[38] KDD Cup 2012. `http://www.kddcup2012.org/`.

[39] Kryo library. `http://code.google.com/p/kryonet/`.

[40] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for Log Processing. In *USENIX ATC*, 2011.

[41] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny Aggregation Service for Ad-hoc Sensor Networks. *OSR*, 36(SI), 2002.

[42] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, 2010.

[43] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[44] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. In *VLDB*, 2010.

[45] Palo Alto Networks. `http://www.paloaltonetworks.com`.

[46] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A Cost Comparison of Data Center Network Architectures. In *CoNEXT*, 2010.

[47] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *SIGCOMM*, 2013.

[48] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, 2013.

[49] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[50] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[51] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem. In *SIGCOMM*, 2012.

[52] A. Shieh, S. Kandula, and E. G. Sirer. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets*, 2010.

[53] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[54] Twitter Storm. `http://goo.gl/Y1AcL`.

[55] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, G. Chockler, H. Li, and Y. Tock. Dr. Multicast: Rx for Data Center Communication Scalability. In *EuroSys*, 2010.

[56] Wikipedia Categories. `http://goo.gl/n60jg1`.

[57] Wikimedia Downloads, June 2012. `http://goo.gl/DQFJk`.

[58] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *SIGCOMM*, 2004.

[59] Y. Yu, P. K. Gunda, and M. Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *SOSP*, 2009.

[60] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[61] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *SIGCOMM*, 2012.