Programmable Multi-Dimensional Table Filters for Line Rate Network Functions

Vishal Shrivastav
Purdue University

ABSTRACT

The ability to filter entries in the data plane from a set or table of resources (e.g., network paths, servers, switch ports) based on multi-dimensional policies over stateful resource-specific metrics (e.g., filter paths with utilization < 0.6 and latency < 3us) is critical for several key network functions, such as performance-aware routing, resource-aware load balancing, network diagnosis, security and firewall. However, current generation of programmable switches do not support table-wide stateful filtering at line rate. We present **Thanos**, which augments the existing programmable switch pipeline with support for programmable multi-dimensional filtering over a set of resources. Thanos seamlessly integrates with multi-terabit programmable switch pipelines at nominal chip area overhead. Our evaluation, based on an FPGA prototype and a simulator, shows that policies expressed in Thanos can improve the performance of key network functions by up to 1.7× compared to state-of-the-art.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Hardware** → **Networking hardware**;

KEYWORDS

Programmable Networks; Switch Architecture

ACM Reference Format:
Vishal Shrivastav. 2022. Programmable Multi-Dimensional Table Filters for Line Rate Network Functions. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544266

1 INTRODUCTION

The advent of programmable switches has allowed network switches to become increasingly more stateful, as switches are now capable of storing large amounts of *stateful metrics* (such as congestion along a path, queuing statistics at ports, packet counters for flows) to make intelligent decisions in the data plane with regards to key network functions, such as routing [9], load balancing [8, 14, 18], congestion control [16], network diagnosis [21], policy compliance under failures [29], and security and firewall [27].

One of the key operation that is at the core of several of the aforementioned network functions is the ability to *filter* a resource (e.g., a network path, a load balancing server, or an egress switch port) from a given set of resources, based on certain filtering policy specified over the stateful metrics associated with the resources. Below are some examples.

- **Performance-aware routing [9].** The path to route a packet or flow is chosen based on the current status of the network path metrics. Below is an example.

```
From the set of all paths, select the path
    with delay < d and utilization < u
```

Figure 1: An example routing policy.

- **Congestion-aware load balancing [2, 8].** The path to load balance a packet or flow or flowlet is chosen based on the current congestion along the path [2, 14] or queuing at egress switch port [8]. Examples include:

```
Randomly choose d out of N possible egress ports.
Find the one with the current minimum queue occupancy
 between these d samples and m least loaded samples
from the last time slot, and send packet on that port
```

Figure 2: Load balancing policy used in DRILL [8].

```
From the set of all paths, select the path
          with least congestion
```

Figure 3: Load balancing policy used in CONGA [2].

- **Stateful L4 load balancing [18, 22].** The server to map a new L4 flow is chosen based on the resource availability at the load balancing servers. Below is an example.

```
From the set of all servers, select the server
with cpu utilization < u and available memory > m
      and available network bandwidth > b
```

Figure 4: An example L4 load balancing policy.

- **Data plane diagnosis [21].** A switch is queried to filter over certain packet, flow, or port level statistics maintained at the switch to diagnose certain abnormal network behavior in real time. Below is an example.

```
Filter all switch ports with packet rate > t
```

Figure 5: An example query for network diagnosis.

- **Security and Firewall [11].** A switch blacklists or drops packets from suspicious flows in real time. Below is an example.

```
If the packet rate for an IP destination D is > T,
   filter (and black-list or drop packets) from
          all source IPs sending to D
```

Figure 6: An example firewall policy.

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544266

- **Policy compliance [29].** In multi-tenant environments, switches enforce policies from different tenants while making routing or load balancing decisions. Below is an example.

```
From all available paths, filter the paths not carrying
tenant A's or B's traffic. Choose a path at random from
 the filtered paths to route a new flow from tenant C.
```

Figure 7: An example policy compliant routing.

All the policies mentioned above can be abstracted as a chain of filter operations that implement the filtering criteria specified in the policy to filter a subset of resources from the set of given resources. The filtering criteria could be stateless, *e.g.,* , select a random resource, or it could use one (Figure 3,5,6) or multiple (Figure 1,2,4,7) stateful resource-specific metrics to filter the desired resources. Further, the filter operations could also be chained, where the output of one filter operation is used as the input for the next (Figure 2,6,7). The ability to express rich filtering policies in the switch data plane not only provides the network operators with much more flexibility in terms of how to route, load balance, diagnose, firewall, or enforce tenant-specific policies on their network's traffic, but it also results in much improved network performance (§7). Unfortunately, however, current generation of programmable switches are unable to express the filter policies mentioned above at line rate due to their constrained memory and computation semantics (§2.2).

In that light, we present **Thanos**, which is an enhanced programmable switch architecture that provides the ability to express rich multi-dimensional filter policies over a set of resources at line rate. Thanos extends the current programmable switch pipeline, namely Reconfigurable Match Table (RMT) [5], with a new hardware module for chained multi-dimensional filtering (§3). Thanos's filter module takes as input a multi-dimensional table of resources, where each dimension corresponds to a resource-specific metric. For example, the set of resources could be network paths, and the set of dimensions could be path delay, path utilization, path length, etc. Next, Thanos allows users to program the filter module with a chain of custom filter policies involving one or more dimensions or metrics, to filter a subset of resources that satisfy the programmed policy during runtime (§4). As is the case with RMT, the programmed filter policy cannot be changed during runtime.

Implementing a programmable multi-dimensional filter module in hardware at line rate is challenging. In that regard, Thanos makes three key contributions. First, Thanos proposes a new hardware data structure, called *Sorted Multidimensional Bidirectional Map (SMBM)* (§5.1) that efficiently exploits the spatial parallelism in hardware to enable line rate multi-dimensional filtering. Second, Thanos extends the set of action units in today's programmable switches with two new action units that can implement a wide-range of filter operations (§5.2). In that regard, Thanos both borrows and extends the filter operations from Codd's relational algebra [7]. And finally, Thanos uses the filter action units to design a fully reconfigurable filter pipeline that can express any arbitrary chain of filter operations supported in Thanos at line rate (§5.3).

Based on ASIC synthesis (§6), we show that Thanos's hardware design can run at clock speeds of state-of-the-art multi-terabit switches while incurring nominal chip area overhead. Our evaluation, based on a FPGA prototype (§7.1) and a simulator (§7.2), shows that rich filter policies expressible in Thanos and those that cannot not be expressed on existing programmable switches, can provide performance gains of up to 1.7× compared to state-of-the-art for routing and load balancing network functions.

Overall, Thanos advances the state of programmable data plane by providing a new abstraction for chained multi-dimensional filtering. And while Thanos was primarily designed for network functions, its abstraction is general enough that it opens up the possibility of offloading several other key applications, beyond network functions, to the network's data plane. Examples include online analytical processing [39], graph database queries [15], and multi-dimensional clustering in ML [33]. In §7.2.5 we show the benefits of offloading one such application, namely graph database queries, to the network data plane using Thanos. Thus, Thanos has the potential to not just improve the performance of network functions but improve the state of in-network computing in general.

This work does not raise any ethical issues.

2 MOTIVATION

In this section, we motivate the need for line rate filtering in the data plane, and show that existing programmable switch data plane architectures are unable to implement rich filter policies at line rate, thus motivating the need for Thanos.

2.1 Need for Line Rate Filtering in Data Plane

The stateful metrics associated with the examples in §1 are typically updated either at RTT granularity (*e.g.,* congestion updates along a path in CONGA [2] and HULA [14]), or in some cases even at per-packet granularity (*e.g.,* local queue length updates in DRILL [8]). In the same vein, recent performance-aware routing systems such as Contra [9] update the stateful routing metrics at RTT granularity to react quickly to changes in traffic and failures. Thus, filter policies over such stateful metrics must be implemented in the data plane, as alternatives such as implementing the policies in the control plane cannot operate at such fine grained timescales.

Further, as stated in the above paragraph, certain stateful metrics like local queue length in switches, update at packet-level timescale. In addition to that, several of the filter policies, especially in the context of routing and load balancing, are applied at a flowlet granularity [2, 12, 14, 44], which can be as small as a few packets. In fact, recent studies [20] suggest that flows with size less than 1000 bytes or 1 MTU packet dominate both the number of flows (over 85%) and the number of bytes (over 70% of all network traffic) for certain popular datacenter workloads. Thus, even traditional flow based routing and load balancing policies now need to practically run at per-packet granularity. Given all this, filter policies in the data plane must be able to run at line rate or packet-level timescale.

Unfortunately, state-of-the-art programmable switch data plane architecture, namely Reconfigurable Match Table (RMT) [5], is unable to implement rich filter policies at line rate. Below we discuss this limitation of RMT in detail.

2.2 Limitations of RMT

Reconfigurable Match Table (RMT) [5], including its stateful variants, Domino [25] and FlowBlaze [23], is state-of-the-art for programmable switch data plane architecture. RMT provides two key

stateful primitives, namely *Match tables* and *register arrays*, that one could potentially use to implement multi-dimensional filtering.

Match tables can have multiple attributes, where each attribute typically corresponds to a packet header field. Match tables match those header field values in the input packet against the entries in the table, and if a match is found, the corresponding action is triggered. The match type could be exact (implemented using SRAM) or ternary (implemented using TCAM). However, Match tables are not capable of filtering the entries in the table based on custom filtering policies. Thus, while one could potentially use Match tables to store the resources with their stateful metrics as table attributes, one cannot perform custom filtering on top.

Register arrays provide a better alternative to implement filter policies, as they can be used to implement custom data structures. However, register arrays in RMT are mainly suited to implement simple data structures, such as hash tables. In fact, one can store the resources and their stateful metrics in a register hash table, and implement simple stateless filter policies at line rate, *e.g.,* selecting a resource at random. But when it comes to filtering resources based on stateful filtering criteria, such as filter the resource with minimum metric value or filter all resources with metric value $< x$, as shown in Figure 1–7, register arrays fall short. This is because RMT allows access to at most single entry per register array per packet per pipeline stage (per clock cycle). As a result, in the worst-case one would need $O(N)$ clock cycles or pipeline stages to iterate through the set of resources for a single filter operation, where N is the number of resources, thus precluding line rate processing. And things would only get worse for policies that have a chain of filter operations.

One could potentially improve on the time complexity of filter operations by trying to implement more sophisticated data structures using register arrays, such as a B-tree [30], known to be efficient for several filter operations. A straight-forward implementation would store each node in B-tree in a register, and map each level of the B-tree to a different pipeline stage in RMT. But even this is hard to implement in RMT, because RMT pipelines are feed-forward, *i.e.,* packets and state can only move forward through the pipeline. As a consequence, one cannot go back and update nodes in level i of the tree once they have traversed to a level $>i$, something which is needed in a B-tree implementation. One possible work-around to access a state in stage i from stage i+1 is to let the packet move forward through the pipeline till the last stage, and then re-circulate the packet to the start of the pipeline to eventually reach stage i. But this would effectively reduce the packet processing throughput of the switch, while also incurring high latency penalty.

Overall, RMT pipelines are not designed to support complex stateful processing. Quoting a line (paraphrased) from Domino [25], *"....RMT/Banzai atoms (action units) model small operations that occur on every packet, and are unsuitable for large operations such as scanning a table that span many clock cycles"*. Hence, in order to implement rich stateful network functions on programmable switches, one must look beyond RMT and design a custom hardware module that could be easily integrated with RMT. This is already the case with network functions such as packet scheduling that fundamentally requires a priority queue for its operation which RMT cannot support. As a result, there have been several custom hardware architectures [4, 24, 26] proposed to augment RMT with a priority queue data structure for scheduling. In the same spirit, in this paper we make the case for augmenting RMT with a new hardware module for multi-dimensional filtering, that can be easily integrated with the RMT pipeline (§3) and has the potential to improve several key network functions (§7.2.2, §7.2.3, §7.2.4), including routing, load balancing, network diagnosis, and firewall, and even certain key distributed applications, such as relational and graph database queries (§7.2.5).

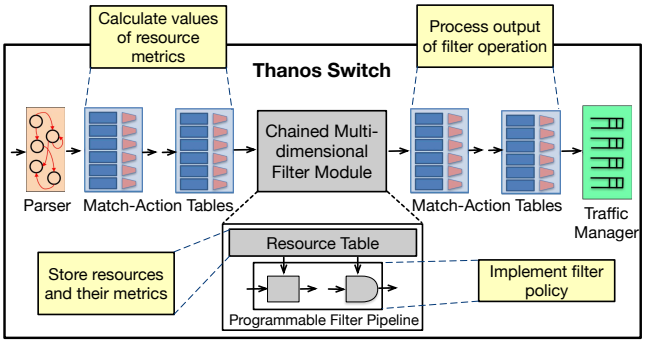

Figure 8: Thanos switch architecture. It also highlights the division of labor between RMT's modules (programmable parser and Match-Action tables) and Thanos's filter module, for implementing custom filter policies.

3 SYSTEM ARCHITECTURE

Figure 8 shows the high-level architecture of Thanos, where the filter module is integrated inline with the Match-Action stages of the RMT pipeline. In general, one can have multiple such filter modules integrated with the RMT pipeline, where each module would express filter policies on a different set of resources.

The filter module is triggered every time a packet arrives at the module. The packet traverses through the filter module unmodified, while, in parallel, the programmed filter policy is applied on the set of resources. The output of the filtering operation is written to packet's metadata for further processing in the RMT stages that follow the filter module. The filter module is fully pipelined, and hence can serve a new packet every clock cycle. Finally, packets that do not want the filtering policy applied to them can simply bypass the filter module altogether.

Next, implementing filter policies over a set of resources involves four key tasks. Figure 8 illustrates the four tasks and how they are divided amongst Thanos's filter module and the RMT pipeline.

(1) **Calculate resource metric values.** This depends on whether the resource metrics are generated remotely (e.g., congestion along a path, resource consumption at a web server) or locally at the switch (e.g., packet counters, queuing at switch ports).
- **Remote metric.** Thanos relies on *probe packets* from remote resources to carry the metric information. This is a standard practice in most switch-based stateful network functions [2, 9, 14]. Further, Thanos relies on the RMT pipeline to process the probe packets and extract the metric information from the header of probe packets. RMT pipelines are well suited for this task as header processing is indeed their primary job.
- **Local metric.** In case the resource metrics are generated locally, there is no need for probe packets to calculate the

metric values. Instead, Thanos relies on basic RMT primitives such as counters, meters, registers, and ALUs to compute simple local metrics such as packet counters. For more complex local metrics, such as those that involve the traffic manager, Thanos leverages the recently proposed *event-driven* packet processing architecture [10] for RMT. This architecture allows for custom events to trigger custom processing inside the RMT pipeline at line rate. An example of this would be incrementing a register value in the RMT pipeline every time an event of packet enqueue into an egress queue happens, and similarly decrementing the register value every time an event of packet dequeue from the egress queue happens. Using this example, one can easily maintain the queue length metric for each egress queue at line rate, such as used in DRILL [8].

(2) **Store resources and their metrics.** This involves designing a data structure to store the resources and their respective metrics. The key requirement from such a data structure is to allow the implementation of rich filtering policies at line rate. As discussed in §2, RMT pipelines do not support such data structures. Hence, in §5.1, Thanos proposes a new hardware data structure for storing resources and their stateful metrics.
(3) **Implement filter policy.** Thanos proposes a new custom filter pipeline (§5.3) and action units (§5.2) that is customized for the resource data structure mentioned above, and together they can implement any arbitrary chain of filter operations at line rate.
(4) **Process output of filter operation.** For most network functions, this involves using the output of the filter operation to make a forwarding decision on the current packet. For example, for routing and load balancing applications, this could involve forwarding the packet along the filtered path, where as for firewall application, this could involve dropping the packet. RMT pipelines are well suited for making such forwarding decisions, as this is again part of their primary job, and hence Thanos maps this final task to the RMT pipeline.

4 ABSTRACTIONS AND PRIMITIVES

This section describes the basic abstractions and primitives underpinning Thanos's filter module. The key insight Thanos uses is that the resources and their corresponding metrics can be represented as a *relational table*, and filter policies can be reduced to chains of filter operations over a relational table. Thus, given N resources each with M metrics, Thanos stores the resources in a global relational table with N entries and M+1 attributes, where one of the attributes is the (unique) resource id (which also acts as the primary key of the table), and the remaining M attributes are the M resource metrics. The filter operations on the table are inspired from the classic Codd's relational algebra operators [7] as described below.

4.1 Basic Filter Operators

Thanos supports two classes of filter operators—(i) unary operators, that filter based on zero or one table attribute, and (ii) binary operators, that merge the outputs of two unary filter operations to support multi-dimensional filtering.

4.1.1 Unary Filter Operators. These operators take as input a single relational table, and filter based on at most one table attribute to return a subset of table rows (or resources) as the output relational table. The operators are chosen based on the following insights. Most network functions filter the resources in one of three ways—(i) based on some *predicate* over an attribute [11], or (ii) based on some *ordering* of an attribute's values, such as a (weighted) round-robin order [47] or some priority order [2, 14]. or (iii) *randomly* [35]. Based on this, Thanos includes five unary filter operators as described below. Note that these operators both borrow from (*e.g.,* predicate and min/max operators) and extend (*e.g.,* round-robin and random operators) the classic unary operators from Codd's relational algebra [7]. At the same, they also exclude Codd's other classic unary operators, namely project and rename, as these operators are not involved with filtering table row entries.

(1) **no-op** ($table_1$): It bypasses all the filtering operations, and simply copies the input table table_1 into the output table.
(2) **predicate** ($table_1$, *attrX* **rel_op** *val*): It operates on the attribute attrX of table_1, and outputs a new table comprising entries whose attrX value satisfy the specified predicate. Here val ∈ Integer and rel_op is a relational operator ∈ $\{<, >, \leq, \geq, ==, \neq\}$.
(3) **min/max** ($table_1$, *attrX*): It outputs a table comprising a single entry from table_1 whose attrX value is the lowest/highest amongst all the entries in table_1.
(4) **round-robin** ($table_1$, *attrX*): It outputs a list comprising a single entry chosen cyclically from table_1 in proportion to the entry's "weight", where the value of attrX is used as a entry's weight.
(5) **random** ($table_1$): It outputs a table comprising a single entry from table_1 chosen uniformly at random.

4.1.2 Binary Filter Operators. These operators take as input two relational tables, each of which is an output of some unary filter operation, and merge them into a single relational table. The binary operators are based on the set operators from the classic relational algebra. Note that Thanos does not include *join*, which is the other classic binary relational algebra operator. This is because Thanos has a single global relational table, and all the binary operations are performed over the subsets of the same global table. Hence, the relational tables used in binary operations always have the same set of attributes, thus making join irrelevant.

(1) **no-op** ($table_1$, $table_2$, *choice*): It bypasses all the merge operations, and simply copies one of the input tables, chosen based on the value of *choice*, into the output table. Thus this operation behaves like a 2:1 MUX.
(2) **union** ($table_1$, $table_2$): It outputs the table with all the entries from both table_1 and table_2.
(3) **intersection** ($table_1$, $table_2$): It outputs the table with entries common to both table_1 and table_2.
(4) **difference** ($table_1$, $table_2$): It outputs the table with entries in table_1 but not in table_2.

4.2 Abstractions for Filter Chaining

In this section, we describe the abstractions for chaining basic unary and binary filter operators to express richer filter policies.

4.2.1 Parallel Chaining. A parallel chain in Thanos is a single linear chain of k unary filter operators, with a single input I_1 and a single final output O, as shown in Figure 9. Each of the k unary filter operators in the chain are identical. The inputs and outputs are relational tables. Each operator i in the chain takes as input the

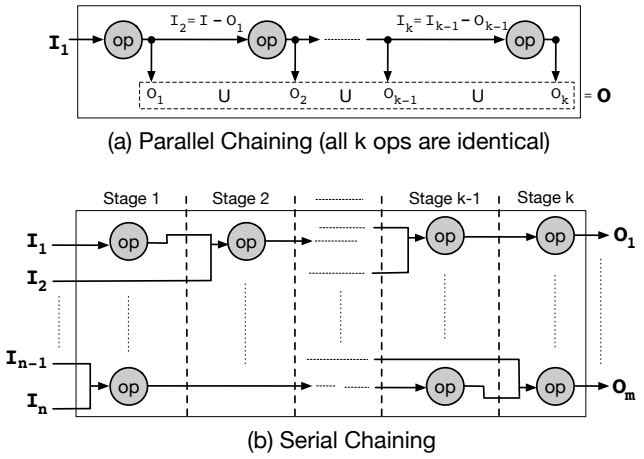

Figure 9: Parallel and Serial Chaining of Operators.

table I_i and produces the output table O_i. The input to operator i, I_i, is the set difference of the input and the output of operator $i-1$.

$$\begin{aligned} I_i &= I_{i-1} - O_{i-1} \quad if\ i > 1 \\ &= I \quad if\ i = 1 \end{aligned} \tag{1}$$

And the final output O is the *union* of outputs of each operator in the chain, *i.e.*, $O = \bigcup_{i=1}^{i=k}\{O_i\}$.

Parallel chaining can be used to express policies [8, 19] that require filtering k min/max elements from a list (via a chain of min/max operators), or filtering k unique random elements from a list (via a chain of random operators).

4.2.2 Serial Chaining. Serial chaining is the more general form of chaining in Thanos, and applies to both unary and binary filter operators. A serial chain takes in n relational tables as input and produces m relational tables as output. The set of operators in any serial chain can be partitioned into a series of k *stages* as shown in Figure 9. We call k the *length* of the serial chain. The set of operators within a single stage do not have input-output dependencies, *i.e.*, an input to an operator within a stage cannot be an output of some other operator from the same stage. Instead, the inputs to stage i must either be one of the n original inputs, or it must be an output from some operator in stage $j < i$. Serial chaining does allow for *output fan-out*, *i.e.*, the output of an operator can serve as input to multiple operators. The outputs from the k^{th} stage are the final outputs of serial chain.

4.2.3 Conditional Policies. Thanos can also express conditional policies of type, ***if*** *(predicate)* ***then*** *policy1* ***else*** *policy2*, by using the parallel and/or serial chain abstractions to express policies *policy1* and *policy2*, and using a MUX at the end to choose between the outputs of those policies based on the predicate. The MUX can be implemented in a single RMT pipeline stage [25] that immediately follows the filter module in Thanos's pipeline.

5 HARDWARE DESIGN

In this section, we describe the hardware design of Thanos's filter module. As shown in Figure 8, the module comprises a resource table, stored as a relational table, and a programmable filter pipeline that performs the filtering operations on the resource table using the abstractions described in §4.

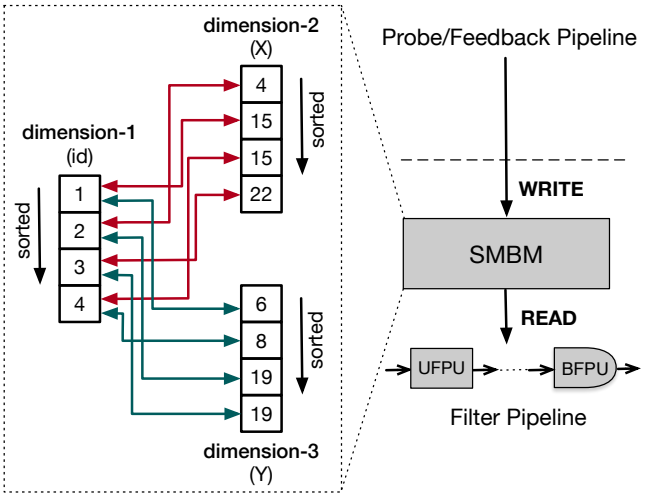

Figure 10: A sorted multidimensional bidirectional map (SMBM) storing four example resources, each with a unique resource id and two stateful metrics X and Y. The figure also shows the processing pipelines SMBM interfaces with.

Design goals. The goal of our hardware design is to have a fully pipelined design that can process a new data packet every clock cycle, while incurring only a small, and more importantly, deterministic processing latency. This would ensure line rate processing to match the processing rate of RMT pipeline for seamless integration.

5.1 Resource Table

Thanos stores the resource table as a relational table using a new hardware data structure called *Sorted Multidimensional Bidirectional Map (SMBM)*, as shown in Figure 10. Assuming N resources and M metrics, SMBM stores the M+1 relational table attributes (resource id + M metrics) as a M+1-dimensional *map* data structure, where each dimension has N entries, one per resource, stored as a *flat* sorted list. The lists are sorted in increasing order, and if any two entries in the list have the same value, the entry that was enqueued first appears first. The first dimension is the resource id (primary attribute), and the remaining M dimensions correspond to the M metrics. Finally, the data structure maintains a bidirectional mapping between the resource id dimension and each of the metric dimensions, *i.e.*, the id of each resource maps to each of its M metrics, and each of the M metrics of a resource map back to its id.

5.1.1 Why SMBM?. The need for a new data structure arises from two key limitations of traditional data structures for filtering.

Lack of a universal data structure. The unary and binary filter operations in Thanos have been very well-studied, but in isolation. As a result, there are different data structures to efficiently execute different operations, including search trees [30], range filters [36], heaps [31], linked lists [38], and disjoint set data structures [34]. Lack of a universal data structure means one has to either compromise on the performance of certain operations, or maintain a separate data structure for each operation, which would mean replicating the state, thus resulting in significant memory overhead plus the overhead of keeping the state synchronized across all replicas.

Performance limits of classic data structures. Most of the classic data structures designed for specific filter operations have one thing in common—they all share a hierarchical structure, *e.g.,* B-tree [30], K-d tree [36], binary heap [31], etc. This fundamentally limits the latency of operations to $O(log(N))$ computation steps on average, with the worst-case performance as bad as $O(N)$ [36]. Thus, these data structures would add high and non-deterministic latencies per operation on the network data path. Further, several of these data structures are also hard to pipeline in hardware [46], thus also limiting the processing throughput. These limitations are further exacerbated when we have chains of these operations. Fundamentally, these classic data structures were designed for CPU-like architectures, and hence fail to fully exploit the available spatial parallelism on a custom hardware.

The SMBM data structure is designed specifically to support fast filter operations in hardware. This is achieved via four distinct features of the data structure. (i) Unlike the classic hierarchical data structures, the *flat* list structures in SMBM are well-suited to exploit the spatial parallelism in hardware to allow parallel decision making over the entire set of resources in a single computation step. (ii) Inspired by the recent hardware data structures such as PIFO [26] and PIEO [24], SMBM keeps the lists *sorted* that accelerates filter operations that rely on ordering, such as range filters and priority queue operations (min/max). (iii) *multidimensionality*, *i.e.,* storing table attributes as independent dimensions (of sorted lists), allows independent filter operations over each attribute, thus enabling further parallelism. And, (iv) *bidirectional* mappings allow for fast translation between resource ids and corresponding metrics via the forward (id → metric) and reverse map (metric → id) respectively.

5.1.2 *SMBM Implementation.*

SMBM comprises M+1 sorted lists, one per dimension, each of size N (Figure 10). The implementation of sorted lists is based on the design described in PIFO [26], where N elements are stored in a N-ported memory, usually implemented using N flip-flops. The use of flip-flops instead of SRAM (used to implement registers in RMT), allows one to read and write each entry in the list in parallel per clock cycle needed for line rate filtering (SRAM typically allows only one read or write per clock cycle). However, this comes at the cost of scalability. This trade-off is further discussed in §6. In addition, each entry in the resource id list also stores M pointers, one per metric, and each entry in each of the M metric lists store a pointer to their corresponding resource id in the resource id list. SMBM allows two primitive write operations,

add(SMBM, id, [metric1: val1, ..., metricM: valM]). This operation adds a new entry with resource id *id* and metric values *val1, ..., valM* to SMBM.

delete(SMBM, id). This operation deletes a SMBM entry with resource id *id*, if present.

The two primitive operations can be combined to do more complex operations, *e.g.,* to *update* the metric values of an existing resource, one can issue a delete operation (which deletes the resource to be modified) followed by an add operation (which re-inserts the resource with updated values).

Both add and delete operations take exactly two clock cycles and are fully pipelined. In the first clock cycle, we search all the SMBM lists in parallel for the appropriate locations where the input entry needs to be added to or deleted from, so that the lists remain sorted even after the operation. Then in the second clock cycle, we add/delete the entries to/from those locations, appropriately shifting the rest of the entries in the lists, while also updating the pointers between the resource id and corresponding metrics. The ability to do list-wide search and shift operations in one clock cycle can be attributed to the parallelism offered by flip flops.

5.1.3 *Latency and Throughput.*

The latency of both write operations is two clock cycles. The design is fully pipelined, and can serve a new write request every clock cycle. Further, SMBM shares a read interface with the filter pipeline (Figure 10). To meet our design goal of processing one packet every clock cycle, SMBM must support a read operation every clock cycle. Fortunately, given the lists in SMBM are stored independently in separate flip flops, one can read the entire SMBM data structure every clock cycle, while also doing writes in parallel.

5.1.4 *Concurrency and Consistency.*

Read and write happen concurrently in SMBM. Since writes take two clock cycles, while reads need to be supported every clock cycle, there is a potential of reading the data structure in an inconsistent write state if it is also concurrently being written. However, this is not an issue in SMBM, since even though write operations take two clock cycles, the data structure is written atomically, since any write to the data structure happens only during a single clock cycle (the second clock cycle for both add and delete operations).

5.1.5 *Integration with multi-pipelined data planes.*

Modern switch data planes are multi-pipelined to achieve higher aggregate processing rates [41]. Thanos's filter module can naturally scale to multiple pipelines by having one separate module per pipeline. However, this would require synchronizing the writes across multiple SMBM replicas. Note that for local switch metrics where SMBM is updated via event-driven processing (§3), this is not an issue as the event will be triggered synchronously on each pipeline. However, for remote metrics where SMBM is updated via probe packets (§3), on a typical RMT pipeline, one would need to recirculate the probe packet through each pipeline, so that the metric update is applied to all replicas. This solution comes with obvious throughput penalty. Instead, in Thanos, if a probe packet in pipeline i is updating SMBM resource entry e, we synchronously update the SMBM entry e in each pipeline, thus not needing re-circulation. The use of flip-flops allows updates issued from multiple pipelines to happen in parallel at each SMBM replica. However, this design could potentially result in a write contention if there was another probe packet in pipeline k ($\neq i$) updating SMBM entry e in the same clock cycle. Fortunately, one can easily avoid such contentions from happening in practice by making sure that the probe packets corresponding to the same resource take the same network path (as is the norm with TCP flows). This would ensure that at any given switch the probe packets for the same resource will always arrive on the same port (and hence processed by the same pipeline), thus precluding the possibility of two probe packets updating the same resource entry simultaneously on any given pipeline.

5.2 Programmable Filter Units

This section describes the design of the two basic filter processing units in Thanos, namely *Unary Filter Processing Unit (UFPU)* and *Binary Filter Processing Unit (BFPU).*

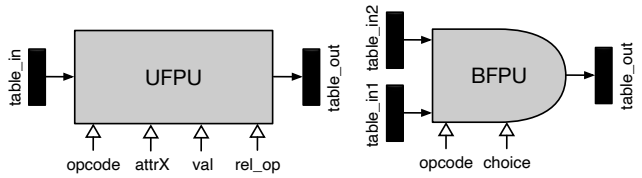

Figure 11: Two basic filter processing units in Thanos—Unary Filter Processing Unit (UFPU) and Binary Filter Processing Unit (BFPU). §4.1 has description of inputs and outputs.

5.2.1 *Unary Filter Processing Unit (UFPU).*

UFPU implements the unary filter operations described in §4.1.1. The processing latency is two clock cycles and the design is fully pipelined.

Input and Output. The inputs and outputs to the UFPU are shown in Figure 11. The *opcode* value is used to configure the choice of UFPU operation. One key design choice we make is in the way we implement the input and output tables. Logically, input and output to a UFPU is the resource table, *i.e.,* a SMBM data structure. However, given that we will have multiple UFPUs in the filter pipeline, passing multiple instances of the entire SMBM data structure through the pipeline is not efficient. Hence, we encode the input and output to UFPUs as pointers to the SMBM entries (implemented as a *bit vector*—the vector is indexed by resource ids, and a value of 1 for index i indicates the existence of resource with id i).

UFPU implements the following unary filter operations.

no-op. In parallel $\forall i$, out_bit_vector[i] = inp_bit_vector[i].

predicate(inp_bit_vector, attrX *rel_op* val). Here rel_op $\in \{<, >, \leq, \geq, ==, \neq\}$. In the first clock cycle, we leverage the flip-flops in SMBM to copy all the entries in the attribute attrX list in parallel into a temp_list. Note that not all entries in the attrX list would be valid, as entries corresponding to resources not present in the inp_bit_vector are invalid. To mask the invalid entries, we leverage the bidirectional mapping in SMBM, using the reverse map to set the entries in temp_list corresponding to an invalid resource as NULL, as shown in the combinational logic below,

temp_list[i] = if (inp_bit_vector[attrX_list[i]→id_ptr])
then attrX_list[i] else NULL

In the second clock cycle, we apply the input predicate to each valid entry in temp_list, and again leverage the SMBM reverse mapping to set the out_bit_vector entry to 1 for each resource that satisfies the predicate. The combinational logic for this operation is,

out_bit_vector[temp_list[i]→id_ptr] =
if ((temp_list[i] ≠ NULL) & (temp_list[i]→val *rel_op* val))
then 1 else 0

min/max(inp_bit_vector, attrX). In the first clock cycle, we copy the attribute attrX list into a temp_list as described above. Since we keep each metric list in SMBM sorted, the very first/last entry would be the min/max entry. Unfortunately however, not all entries in the temp_list are valid entries, as the entries corresponding to resources not present in the input list are masked out. Hence, we need to find the first/last *valid* entry in the temp_list as our output. We do this in the second clock cycle, where we first create a bit vector by using the parallel comparison (temp_list[i] ≠ NULL), and feed the bit vector to a priority encoder that returns the index idx of the first (or last) 1 in the bit vector, which would correspond to the first (or last) valid entry in the temp_list, which, in turn, would correspond to min (or max) entry. We set out_bit_vector[idx] to 1 with rest of its entries set to 0.

round-robin(inp_bit_vector, attrX). UFPU maintains an internal state of <last_id, w>, where last_id is the last resource id selected in the round-robin order, and w is the number of times in a row last_id has been selected. In the first clock cycle, starting from the index last_id, we find the next valid index i in the inp_bit_vector, *i.e.,* the next index in cyclic order set to 1, by feeding the bit vector

{inp_bit_vector[last_id : N-1], inp_bit_vector[0 : last_id-1]}

to a priority encoder. In parallel, we also copy the attribute attrX list into a temp_list. In the second clock cycle, we check whether the predicate (inp_bit_vector[last_id] == 1) is true and w is less than or equal to *weight*, where *weight* is the value of attribute attrX corresponding to resource with id last_id. If true, we select resource with id last_id as output by setting out_bit_vector[last_id] to 1 and incrementing w by 1. Else, we select resource with id i (from the first clock cycle) as the output by setting out_bit_vector[i] to 1 and updating the internal state to <i, 1>.

random(inp_bit_vector). Assuming inp_bit_vector of size N, in the first clock cycle, we generate a random number r between 0 and $N-1$ using a standard random number generator such as Linear-feedback shift register (LFSR) [37]. This gives us a random index r into the inp_bit_vector. In the second clock cycle, we check if (inp_bit_vector[r] == 1). If so, we set the out_bit_vector[r] to 1 (with rest of its entries set to 0), else we find the first index i set to 1 in bit vector {inp_bit_vector[r : N-1], inp_bit_vector[0 : r-1]} using a priority encoder, and set the out_bit_vector[i] to 1 (with rest of its entries set to 0).

5.2.2 *Binary Filter Processing Unit (BFPU).*

BFPU implements the binary filter operations described in §4.1.2. The processing latency is exactly one clock cycle.

Input and Output. The inputs and outputs to the BFPU are shown in Figure 11. Similar to UFPU, *opcode* value is used to configure the choice of BFPU operation, and the input and output tables are encoded as bit vectors.

BFPU implements the following binary filter operations.

no-op. It implements the following combinational logic:

out_bit_vector = if (choice == 0)
then inp_bit_vector_1 else inp_bit_vector_2

One of the benefits of storing the tables as bit vectors is that it reduces the set operations in BFPU to simple bitwise logic operations over the two input bit vectors, which can be implemented within a clock cycle in the hardware.

union: inp_bit_vector_1 *bitwise OR* inp_bit_vector_2
intersection: inp_bit_vector_1 *bitwise AND* inp_bit_vector_2
difference: inp_bit_vector_1 *bitwise AND* ~inp_bit_vector_2

5.3 Programmable Filter Chain Pipeline

This section describes how the basic processing units from §5.2 can be used to design programmable parallel and serial operation chains from §4.2, that can express any arbitrary chain of filter operations.

5.3.1 *Programmable Parallel Chain Pipeline.*

Parallel chain pipeline implements the parallel operation chaining abstraction

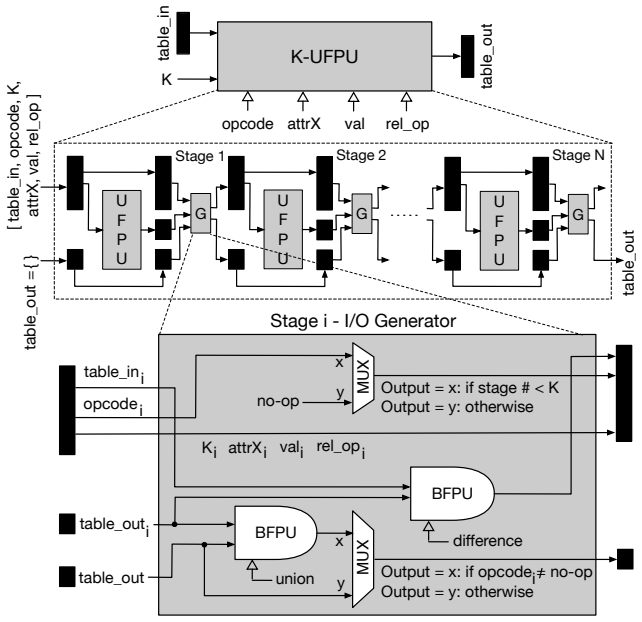

Figure 12: Programmable parallel chain pipeline.

as described in §4.2.2. Figure 12 shows the pipeline design. The pipeline is a linear chain of N UFPUs. We call this K-UFPU. The interface to K-UFPU is the same as the UFPU, except for one extra input called K, which specifies the number of UFPUs (out of N) to be programmed with the operation specified in the opcode. The K programmed UFPUs are always the first K UFPUs in the pipeline closest to the input, while the last remaining $N-K$ UFPUs are programmed with the opcode no-op and simply act as a bypass circuit that has no effect on the final output. The inputs to each UFPU in the chain are generated according to Equation 1, implemented using a series of *I/O generators* shown in Figure 12. Note that by setting K=1, K-UFPU becomes functionally equivalent to UFPU. Finally, our design is fully pipelined, assuming the implementations of its building blocks, UFPUs and BFPUs, are fully pipelined.

5.3.2 Programmable Serial Chain Pipeline. Serial chain pipeline implements the serial operation chaining abstraction as described in §4.2.2. Abstractly, a serial chain pipeline with n inputs, maximum output fan-out of f, and k stages could express any arbitrary serial chain of operations of length at most k, subject to a maximum output fan-out of f and a maximum of n (out of nf) inputs active per stage. An operation could either be a unary filter operation (including parallel chains of unary filter operations), or a binary filter operation. These operations are expressed using K-UFPUs (that can express both basic unary filter operations as well as parallel chains of such operations) and BFPUs.

Figure 13 shows the programmable pipeline for serial chaining. Each stage in the pipeline operates on n active input lines to produce n outputs. Output from each stage has a fan-out of f, thus resulting in nf input lines to each stage except the first, out of which a maximum of n input lines can be active at a given time.

Unlike the parallel chain pipeline (K-UFPU), that has a single input and output with homogeneous processing units (UFPUs), the serial chain pipeline has n (active) inputs and outputs per stage

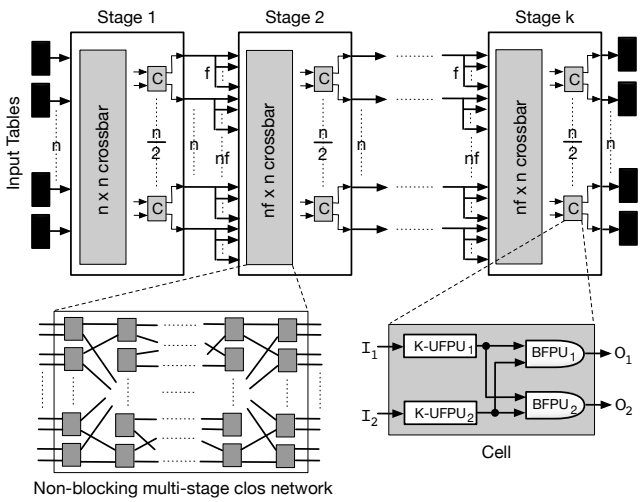

Figure 13: Programmable serial chain pipeline.

with heterogeneous processing units (both UFPUs and BFPUs). This makes it challenging to design a programmable or *fully reconfigurable* pipeline, i.e., a pipeline that could be configured to express arbitrary serial chains of filter operations. To achieve full reconfigurability, *each* pipeline stage with n active input and output lines must satisfy the following property,

Full Reconfigurability. *The ability to apply any of the x K-UFPU operations to any of the n input lines and any of the y BFPU operations to any of the $n(n-1)/2$ pairs of input lines, and connect the output of any operation to any of the n output lines, subject to condition that an output line connects to at most one operation output.*

A naive way to design a fully reconfigurable pipeline stage is to directly connect the input lines to the inputs of processing units (K-UFPUs and BFPUs) using a *crossbar*, as then one can simply configure the crossbar to connect any input line (or pairs of input lines for binary operations) to any K-UFPU or BFPU. However, due to presence of heterogeneous processing units, for n input lines, one would require n K-UFPUs and $n/2$ BFPUs for full reconfigurability, which, in turn, would result in a crossbar of size $n \times 2n$ ($nf \times 2n$ if we also consider fan-out). This is clearly sub-optimal. An optimal design for n input lines, each with a fan-out of f, should have a crossbar of size $nf \times n$. Achieving optimal crossbar size is critical, as the sub-optimal design has twice the wiring complexity, which would result in increased chip area and signal delays [25, 28]. Next, we describe how to design a pipeline with optimal crossbar size.

Instead of connecting each K-UFPU and BFPU directly to the crossbar as in the naive design, the key insight Thanos uses is to break the connection into two stages—in the first stage, Thanos combines pairs of K-UFPUs and BFPUs to form a larger processing unit, called a **Cell**, and then connects each Cell to the crossbar in the second stage. To scale this design to n input lines, Thanos needs $n/2$ Cells in each pipeline stage, as illustrated in Figure 13. This automatically leads to a crossbar of size $nf \times n$, assuming a fan-out of f. Finally, we show below that the above design is still fully reconfigurable like the naive design.

We start with the Cell. Each Cell comprises two K-UFPUs and two BFPUs, connected using cheap 2×2 crossbar as shown in Figure 13. Given this, it is fairly easy to see that a Cell with 2 inputs and outputs

SMBM	N=64	N=128	N=256	N=512
m=2	0.012 mm^2 4.4 GHz	0.029 mm^2 4 GHz	0.071 mm^2 3.6 GHz	0.186 mm^2 2.9 GHz
m=4	0.020 mm^2 4.3 GHz	0.046 mm^2 4.2 GHz	0.109 mm^2 3.6 GHz	0.267 mm^2 2.5 GHz
m=8	0.036 mm^2 4.9 GHz	0.080 mm^2 3.7 GHz	0.183 mm^2 3.6 GHz	0.425 mm^2 2.5 GHz

Table 1: Clock rates and chip area for SMBM.

	N=64	N=128	N=256	N=512
BFPU	216 um^2 40 GHz	431 um^2 40 GHz	852 um^2 40 GHz	0.002 mm^2 40 GHz
UFPU	0.001 mm^2 3.8 GHz	0.002 mm^2 2.2 GHz	0.005 mm^2 1.9 GHz	0.012 mm^2 1.8 GHz

Table 2: Clock rates and chip area for UFPU and BFPU.

	K=2	K=4	K=8	K=16
Cell	0.016 mm^2 2.1 GHz	0.032 mm^2 2.1 GHz	0.063 mm^2 2.1 GHz	0.126 mm^2 2.1 GHz

Table 3: Clock rates and chip area for a Cell.

	k=2	k=4	k=8
n=2	0.067 mm^2 2.1 GHz	0.131 mm^2 2.1 GHz	0.261 mm^2 2.1 GHz
n=4	0.135 mm^2 2.1 GHz	0.270 mm^2 2.1 GHz	0.545 mm^2 2.1 GHz
n=8	0.281 mm^2 2.1 GHz	0.562 mm^2 2.1 GHz	1.125 mm^2 2.1 GHz

Table 4: Clock rates and chip area for filter pipeline.

is fully reconfigurable. For example, to apply a K-UFPU operation to I_1 and another K-UFPU operation to I_2, and connect their outputs to O_1 and O_2 respectively, one would configure the two K-UFPUs with the desired operation and the two BFPUs, $BFPU_1$ and $BFPU_2$, to a no-op with *choice* values 0 and 1 respectively. Similarly, to apply a BFPU operation to I_1 and I_2, and connect the output to O_1, one would configure both the K-UFPUs to no-op and $BFPU_1$ to the desired binary operation.

Next, to show that the overall design is also fully reconfigurable, we note that we can connect any input line to any Cell via the crossbar, and with each Cell being fully reconfigurable themselves, we can configure each Cell to apply the desired filter operation and connect the output of the operation to any of the two Cell outputs. For example, consider a pipeline stage with n=8. To apply a K-UFPU operation to an input line I_4 and connect it to the output line O_8, one would configure the crossbar to connect I_4 to the first input of 4^{th} Cell, configure K-$UFPU_1$ in the 4^{th} Cell to the desired operation, and configure $BFPU_2$ in the 4^{th} Cell to no-op with *choice* value 0. Figure 14 illustrates how Thanos's pipeline can be configured to express an example filter policy.

Finally, even an optimally sized crossbar ($nf \times n$) can have high wiring complexity and large signal delays as n becomes large. To handle this, instead of using a single large crossbar at each stage, Thanos uses a multi-stage non-blocking switching network, such as a clos network [32], built out of smaller crossbars. Traditionally, the challenge with multi-stage switching networks is to dynamically find disjoint routes through the network to connect various input-output pairs [13]. However, this is only an issue in online settings, *e.g.,* forwarding packets in a network. It is not an issue in Thanos since the crossbars in Thanos are configured at compile time based on the input policy, and configurations do not change at runtime.

Finally, our design is fully pipelined, assuming the implementations of its building blocks, K-UFPUs and BFPUs, are fully pipelined.

Table 5 shows various filter policies expressed in Thanos.

6 CLOCK SPEED AND CHIP AREA

The novelty in Thanos's architecture is the introduction of a new chained multi-dimensional filter module. In this section, we report the ASIC clock speed and chip area overhead for this new module. We synthesized our design in Synopsys Design Compiler tool [42] using an open-source 15 nm process technology [17].

We set the performance goal for clock speed as 1 GHz, which is the clock speed of state-of-the-art multi-terabit switches [25, 26]. For chip area, we note that state-of-the-art switching chips occupy 300—700 mm^2 [6], and our goal is to show that adding Thanos's filter module results in a nominal chip area overhead.

Design Parameters. First we enumerate the various design parameters in Thanos along with their default values used for evaluation.

(1) **Resource list size (N):** This refers to the number of resources. Typical examples of resources include network paths, switch ports, servers in a cluster (*e.g.,* a datacenter rack), etc. We expect N to be of the order of few hundreds for most resource types. We set the default value to 128.
(2) **Num of metrics (m):** This refers to number of stateful metrics associated with a resource. We set the default value to 4.
(3) **Parallel chain length (K):** This refers to the length of K-UFPU. Typically it is used to filter top-K/least-K values or K random values from the resource list. We set the default value to 4.
(4) **Num of pipeline inputs (n):** This dictates the maximum number of operations that one can do in parallel per pipeline stage. We set the default value to 4.
(5) **Fan-out (f):** We set the default value to 2.
(6) **Number of pipeline stages (k):** This dictates the maximum length of a serial chain that the pipeline can support. We set the default value to 4.

We choose the default values of the parameters with an understanding that these values can support most practical network filter policies, such as the ones shown in Table 5.

Table 1—Table 4 show the clock speeds and chip area for various building blocks in Thanos's filter module as function of different parameters. If we focus on the overall filter pipeline, then an n-input k-stage pipeline comprises k crossbars and $n/2{*}k$ Cells (Figure 13). Table 4 shows the clock rate and chip area for filter pipeline as a function of n and k. We implemented the crossbar in each pipeline stage using a special multi-stage clos network, called Benes network [13]. The area consumed by the pipeline increases linearly with both n and k. Also, the total area consumed is dominated by the Cells, that account for over 90% of the total pipeline area. The clock rate for the entire pipeline is the same as that of an individual Cell, which, in turn, is the same as that of an individual UFPU inside the Cell. Hence, the clock rate of filter pipeline is independent of n and k and only depends upon the clock rate of an individual UFPU. Finally, for perspective, even a 8×8 pipeline results in an overhead of just 0.3—0.15% in terms of the chip area and can run at twice the clock rate of state-of-the-art switching chips.

Policy	Thanos Filter Chain
	Resource Table → K-UFPU → Resource table; Resource Table, Resource Table → BFPU → Resource Table
Policy 1 in §7.2.3 (*e.g.,* ECMP[35])	K=1, random
Policy 2 in §7.2.3 (*e.g.,* CONGA[2])	K=1, min(util)
Policy 2 in §7.2.2	K=1, cpu < X; K=1, mem > Y; K=1, bw > Z; intersection; intersection; K=1, random; K=1, random; MUX
Policy 3 in §7.2.3	K=X, min(queue); K=X, min(loss); K=X, min(util); intersection; intersection; K=1, min(util); K=1, min(util); MUX
Policy 3 in §7.2.4 (DRILL[8])	K=d, random; previous m samples; union; K=1, min(queue); K=m, min(queue)

Table 5: Expressing example filter policies in Thanos.

Scalability vs. Performance Trade-off. As discussed in §5.1.2, Thanos uses flip flops instead of SRAM for SMBM implementation for performance. However, use of flip flops comes at the cost of scalability. In particular, Thanos is not able to operate at 1 GHz clock speed beyond few 1000s of resources. We note that this is still more than sufficient for most resource types, e.g., switch ports, or network paths, or servers within a cluster (*e.g.,* datacenter rack).

7 EVALUATION

In this section, we evaluate the performance of Thanos using an FPGA prototype and a packet-level simulator.

7.1 FPGA Prototype

We implemented the entire Thanos pipeline shown in Figure 8 in System Verilog [40] and synthesized it on NetFPGA Sume from Xilinx [43], that has four 10 Gbps ports. We used the open-source P4FPGA [45] code to borrow the implementation of RMT pipeline, and extended it with our implementation of Thanos's chained multi-dimensional filter module described in §5.

7.2 Performance Evaluation

In this section, we evaluate the performance benefits of expressing rich filter policies in Thanos for various network functions, as well as for a graph database query application.

7.2.1 Testbed and Simulator. Our testbed comprises six FPGAs, each implementing Thanos switch pipeline as described in §7.1. The FPGAs are connected in a leaf-spine topology as shown in Figure 15, and have eight Dell servers attached to them. Each server is running Ubuntu 14.04 operating system with TCP, and comprises 2 CPUs running at 2.4 GHz, 4 GB of RAM, and 10 Gbps access link bandwidth.

For larger scale network experiments, we use a packet-level simulator to build a network with ~450 hosts connected using a FatTree [1] topology. All link bandwidth is 10 Gbps.

7.2.2 L4 load balancing over end servers. First we consider layer-4 load balancing over end servers (resources). We use a graph database hosted on our university's production servers as the case study. The database is replicated over multiple servers for scaling.

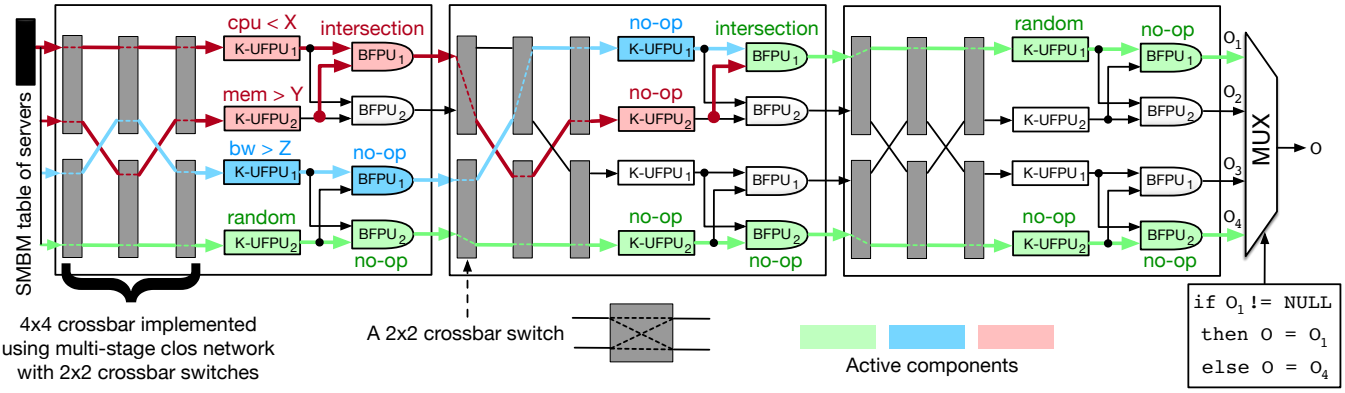

Figure 14: An example to demonstrate how a filter policy is mapped to Thanos's hardware pipeline. In this example we express Policy 2 in §7.2.2 (ref. Table 5) using Thanos's pipeline. The example pipeline comprises 3 stages and 4 inputs and outputs with fan-out value of 1. Each stage has 2 Cells and one 4x4 crossbar. The input to the pipeline is the SMBM table storing the list of load balancing servers and their corresponding stateful metrics (memory, cpu, and bandwidth utilization). The outputs from Thanos's pipeline are MUXed in a RMT match-action stage to get the final output. The diagram also shows the configuration of each active 2x2 crossbar switch and the opcode of each active K-UFPU and BFPU in each Cell. As mentioned in §5.3.2, these configurations are set at compile time for each given filter policy.

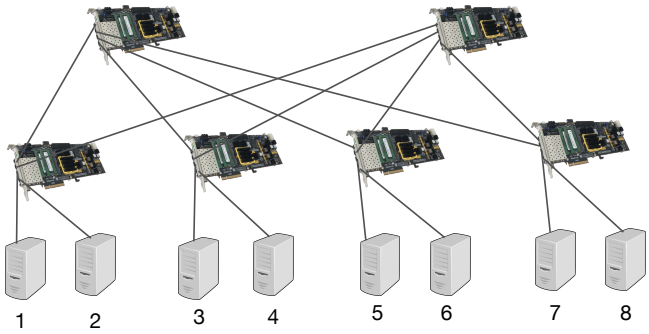

Figure 15: Testbed comprising six FPGAs as switches (§7.1), connected to eight hosts. All links are 10 Gbps.

As is common, each server also hosts other services for resource and cost efficiency via statistical multiplexing. We benchmark a subset of servers for a week in terms of how server resources available to the graph database change over time, and also capture an (anonymized) trace of queries made to the database over the course of the week. Next, we implement a toy version of the graph database on four (5–8) hosts in our testbed, and emulate all other applications that each server runs using a background process that consumes resources in accordance with our resource consumption benchmark. The remaining four hosts (1–4) act as clients generating the queries based on the trace obtained from our benchmark.

We also add the probing functionality where each server periodically generates probe packets with its current resource consumption (CPU utilization, available memory and bandwidth), and sends them to the two spine switches. The spine switches parse the probe packets and extract the metric values as explained in §3. Finally, to ensure connection affinity for L4 connections, we implemented a basic version of SilkRoad [18] with a single exact-match key-value table storing the mappings between the connection/flow ids and the servers. We did not implement advanced SilkRoad functionalities that take care of network dynamics, such as failures, node addition and removal, etc. The spine switches run one of the following load balancing policies,

Policy 1. Select a server uniformly at random.

Policy 2. Select a server uniformly at random from the set of servers with CPU utilization $< X$ and available memory $> Y$ and available bandwidth $> Z$. If the filtered set is empty, select a server uniformly at random from the entire set.

Policy 1 is the typical policy implemented in most state-of-the-art load balancers [18, 22] (including our university's load balancer), and can be easily implemented on today's programmable switches, *e.g.*, using a hash function. However, policy 1 load balances oblivious to available resources at a server for processing the graph queries, and hence can easily map queries to servers that are running low on resources, while there might be servers available with more computing resources. In policy 2, we do a resource-aware load balancing, by first filtering a set of servers with enough resources available (for our experiments we used X=70%, Y=1 GB, and Z=2 Gbps), and then choosing a server randomly amongst those servers. If this returns null, policy 2 defaults to policy 1.

Figure 16 shows the CDF of the response time for each query for policy 2 normalized w.r.t. policy 1. We keep the network load low, so the response time is not affected by network queuing, and only by processing at the servers. Policy 2 achieves between 1.7×–1.3× better response time for 70% of the queries compared to policy 1. This shows that even a simple stateful policy such as policy 2 that could not be implemented on today's programmable switches could result in significant performance gains.

7.2.3 Performance-aware routing. Next we consider routing over network paths (resources). We configure each of the leaf switches to run one of the following routing policies to load balance over multiple available upstream paths,

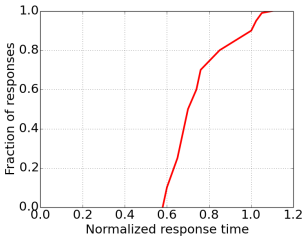

Figure 16: Response time for Policy 2 in §7.2.2 normalized w.r.t. Policy 1.

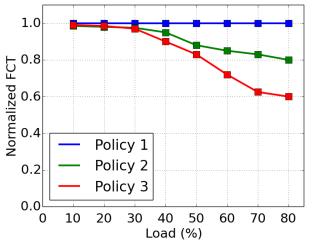

Figure 17: Mean FCT for policies 1, 2, 3 in §7.2.3 normalized w.r.t. Policy 1.

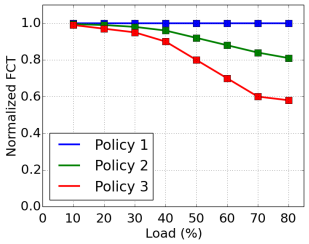

Figure 18: Mean FCT for policies 1, 2, 3 in §7.2.4 normalized w.r.t. Policy 1.

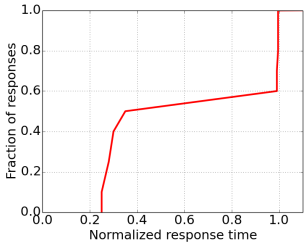

Figure 19: Response time with in-network caching normalized w.r.t. no caching.

Policy 1. Select a path uniformly at random.

Policy 2. Select a path with least utilization.

Policy 3. Filter paths that are simultaneously amongst the top-X paths with least queuing and top-X paths with least loss rate and top-X paths with least utilization, and select the least utilized path from the filtered set. If the filtered set is empty, choose the least utilized path from all available paths.

Each switch periodically generates the queuing, loss rate, and utilization metrics for its links and sends it to all the leaf switches. Each leaf switch parses the probe packets and updates the various metrics for each path, similar to CONGA [2].

We use the Web search [3] traffic trace for experiments. Flows arrive according to a Poisson process and the source and destination for each flow is chosen uniformly at random. We report the mean flow completion times (FCT). Figure 17 shows the simulation results for mean FCT (normalized w.r.t. policy 1) for different network load values (we also ran these experiments on our testbed, but given that we only had 2 upstream paths for each flow, the performance difference was negligible). Policies 1 and 2 are instances of the two most common load balancing policies implemented inside datacenter switches [2, 35]. And yet policy 3, which cannot be implemented on existing programmable switches, achieves the best performance, (1.6× better than policy 1 and 1.3× better than policy 2 at 80% load). This can be attributed to the fact that policy 3 tries to simultaneously optimize for multiple relevant path metrics that could affect the FCT (for our experiments, we used X=5), before defaulting to policy 2 (which only considers one relevant metric).

7.2.4 *Load balancing over switch ports.*

Next, we consider load balancing over switch ports (resources). Each switch in the network runs one of the following load balancing policies,

Policy 1. Select an output port uniformly at random.

Policy 2. Select the least queued output port.

Policy 3. Randomly choose d out of N possible output ports, find the one with the current minimum queue occupancy between these d samples and m least loaded samples from previous time slot, and route packet to that port.

Policy 3 was proposed in a recent system called DRILL [8]. However, due to the limitations of current programmable switches, the authors were not able to implement and evaluate their policy on a real switch. We implement policy 3 in our testbed, and run the same experiments as in §7.2.3. We observe the performance difference between policy 3 (with d=2 and m=1 as suggested in DRILL) and policies 1 and 2 is negligible, which can again be attributed to small number of outgoing ports (N=2) in our testbed. However, Figure 18 shows the mean FCT (normalized w.r.t. policy 1) on our simulator, and here policy 3 outperforms policy 1 and 2 by 1.7× and 1.4× respectively at 80% load, consistent with the numbers reported in DRILL. Interestingly, we note that for our experiments, d=4 and m=4 worked best, which is different from the configuration (d=2 and m=1) suggested in DRILL. We attribute this to different implementation and simulation environment.

7.2.5 *In-network caching of graph filter queries.*

Finally, we highlight the potential of Thanos beyond network functions. We again consider the graph database from §7.2.2. Each node in the graph represents a course, and is associated with certain number of attributes (*e.g.,* course number, term offered, pre-requisites). There is a directed edge between two courses if one course is a pre-requisite of another. Next, based on the (offline) analysis of the captured trace of queries, at each leaf switch, we cache the most popular nodes (courses) in the SMBM data structure, and implement the most popular filter queries using Thanos's filter pipeline. We re-run the same trace from §7.2.2 with spine switches implementing policy 2. Figure 19 shows the CDF of the response time for policy 2 with in-network caching (normalized w.r.t. no caching). The cached queries account for ~50% of all queries, and caching improves the response time for these queries by 4×–2.8×, as it saves both the round trip network delay and processing delay at the server.

8 CONCLUSION

We presented Thanos, which augments the RMT switch architecture with support for chained multi-dimensional filtering over a set of resources. We show that in-network support for such an abstraction can improve the performance of several key network functions, and even general distributed applications, such as graph database queries. Thanos's hardware design could run in excess of 1 GHz on an ASIC while incurring nominal chip area overhead. Our evaluation, based on an FPGA prototype and a simulator, shows that policies expressed in Thanos can the improve performance of key network functions by up to 1.7× compared to state-of-the-art.

ACKNOWLEDGMENTS

I thank the anonymous SIGCOMM'22 reviewers and the shepherd, Ang Chen, for their valuable comments and feedback. I also thank Brent Stephens and all other anonymous NSDI'22 reviewers for their feedback that greatly improved this paper.

REFERENCES

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. *A Scalable, Commodity Data Center Network Architecture.* SIGCOMM.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. *CONGA: Distributed Congestion-aware Load Balancing for Datacenters.* SIGCOMM.

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

[4] Ranjita Bhagwan and Bill Lin. 2000. *Fast and Scalable Priority Queue Architecture for High-Speed Network Switches.* INFOCOM.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. *Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN.* SIGCOMM.

[6] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. *dRMT: Disaggregated Programmable Switching.* SIGCOMM.

[7] E.F. Codd. 1970. *A Relational Model of Data for Large Relational Data Banks.* Communications of the ACM.

[8] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. *DRILL: Micro Load Balancing for Low-latency Data Center Networks.* SIGCOMM.

[9] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. 2020. *Contra: A Programmable System for Performance-aware Routing.* NSDI.

[10] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. 2019. *Event-Driven Packet Processing.* HotNets.

[11] Alexandre Ilha, Ângelo Lapolli, Jonatas Marques, and Luciano Gaspary. 2020. *Euclid: A Fully In-Network, P4-based Approach for Real-Time DDoS Attack Detection and Mitigation.* IEEE Transactions on Network and Service Management.

[12] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. 2007. *Dynamic Load Balancing without Packet Reordering.* HotNets.

[13] Abbas Karimi, Kiarash Aghakhani, Seyed Ehsan Manavi, Faraneh Zarafshan, and S.A.R. Al-Haddad. 2014. *Introduction and Analysis of Optimal Routing Algorithm in Benes Networks.* Procedia Computer Science.

[14] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. *HULA: Scalable Load Balancing Using Programmable Data Planes.* SOSR.

[15] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. 2017. *ZipG: A Memory-Efficient Graph Store for Interactive Queries.* SIGMOD.

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

[17] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. *Open Cell Library in 15nm FreePDK Technology.* ISPD.

[18] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. *SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs.* SIGCOMM.

[19] Michael Mitzenmacher. 2001. *The power of two choices in randomized load balancing.* Transactions on Parallel and Distributed Systems.

[20] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. *Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities.* SIGCOMM.

[21] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2018. *Language-Directed Hardware Design for Network Performance Monitoring.* SIGCOMM.

[22] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. *Stateless Datacenter Load-balancing with Beamer.* NSDI.

[23] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. 2019. *FlowBlaze: Stateful Packet Processing in Hardware.* NSDI.

[24] Vishal Shrivastav. 2019. *Fast, Scalable, and Programmable Packet Scheduler in Hardware.* SIGCOMM.

[25] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. *Packet Transactions: High-Level Programming for Line-Rate Switches.* SIGCOMM.

[26] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. *Programmable Packet Scheduling at Line Rate.* SIGCOMM.

[27] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. *Heavy-Hitter Detection Entirely in the Data Plane.* SOSR.

[28] Brent Stephens, Aditya Akella, and Michael Swift. 2018. *Your Programmable NIC Should be a Programmable Switch.* HotNets.

[29] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. 2021. *D2R: Policy-Compliant Fast Reroute.* SOSR.

[30] https://en.wikipedia.org/wiki/B-tree. 2021. *B-tree.* Wikipedia.

[31] https://en.wikipedia.org/wiki/Binary_heap. 2021. *Binary Heap.* Wikipedia.

[32] https://en.wikipedia.org/wiki/Clos_network. 2021. *Clos network.* Wikipedia.

[33] https://en.wikipedia.org/wiki/Clustering_high-dimensional_data. 2021. *Clustering high-dimensional data.* Wikipedia.

[34] https://en.wikipedia.org/wiki/Disjoint-set_data_structure. 2021. *Disjoint-set datastructure.* Wikipedia.

[35] https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing. 2021. *Equal-cost multi-path routing.* Wikipedia.

[36] https://en.wikipedia.org/wiki/K-d_tree. 2021. *k-d tree.* Wikipedia.

[37] https://en.wikipedia.org/wiki/Linear-feedback_shift_register. 2021. *Linear-feedback shift register.* Wikipedia.

[38] https://en.wikipedia.org/wiki/Linked_list. 2021. *Linked List.* Wikipedia.

[39] https://en.wikipedia.org/wiki/Online_analytical_processing. 2021. *Online analytical processing.* Wikipedia.

[40] https://en.wikipedia.org/wiki/SystemVerilog. 2021. *System Verilog.* Wikipedia.

[41] https://p4.org/assets/p4_d2_2017_programmable_data_plane_at_terabit_speeds.pdf. 2017. *Data Plane Programming at Terabit speeds.* Barefoot Networks.

[42] https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html. 2021. *DC Ultra RTL Synthesis.* Synopsys.

[43] https://www.xilinx.com/products/boards-and-kits/1-6ogkf5.html. 2021. *NetFPGA SUME Virtex-7 FPGA Development Board.* Xilinx.

[44] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. *Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching.* NSDI.

[45] Han Wang, Robert Soule, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. *P4FPGA: A Rapid Prototyping Framework for P4.* SOSR.

[46] Yi-Hua E. Yang and Viktor K. Prasanna. 2010. *High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA.* FPGA.

[47] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. 2014. *WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers.* EuroSys.

A ARTIFACT APPENDIX

Abstract

The artifact includes the source code for each hardware component of Thanos, implemented in System Verilog, along with the 15 nm process technology file used to synthesize the hardware design. The artifact also includes the links to external open source code used to run performance evaluation experiments in the paper (in §7.2).

Scope

The artifact can be used to validate and reproduce the area and timing results for Thanos's hardware design, reported in Table 1, Table 2, Table 3, and Table 4. However, to validate and reproduce the performance evaluation results, one would first need to build the testbed shown in Figure 15, which is out of scope for the artifacts released.

Contents

The artifact contains following folders and files,

(1) **Cell folder.** This folder contains the source code for Thanos's Cell hardware design (in .sv files). It also contains .tcl scripts that can be run to generate the area and timing files for the hardware design. The .area and .timing files in the folder contain the results for one such run of the tcl scripts.
(2) **benes folder.** The folder contains the source code for the crossbar used in Thanos's filter pipeline, implemented using a multi-stage benes network.
(3) **bfpu folder.** This folder contains the source code for Thanos's BFPU hardware design (in .sv files). It also contains .tcl scripts that can be run to generate the area and timing files for the hardware design. The .area and .timing files in the folder contain the results for one such run of the tcl scripts.
(4) **filter_pipeline folder.** This folder contains the source code for Thanos's programmable serial filter pipeline hardware design (in .sv files). It also contains .tcl scripts that can be run to generate the area and timing files for the hardware design. The .area and .timing files in the folder contain the results for one such run of the tcl scripts.
(5) **params folder.** The folder contains the parameter files, along with source code for lfsr and priority encoder modules used in the implementation of UFPU.
(6) **smbm_* folder.** This folder contains the source code for SMBM hardware design (in .sv files). It also contains .tcl scripts that can be run to generate the area and timing files for the hardware design. The .area and .timing files in the folder contain the results for one such run of the tcl scripts.
(7) **ufpu folder.** This folder contains the source code for Thanos's UFPU hardware design (in .sv files). It also contains .tcl scripts that can be run to generate the area and timing files for the hardware design. The .area and .timing files in the folder contain the results for one such run of the tcl scripts.
(8) **NanGate_15nm_OCL_fast.db file.** This file contains the 15 nm process technology information used to synthesize Thanos's hardware design.

Hosting

The artifact is available at https://github.com/vishal1303/Thanos.

Requirements

The hardware design was synthesized using Synopsys Design Compiler RTL Synthesis version L-2016.03-SP2 for area and timing results. The testbed used for performance evaluation results (Figure 15) comprised six NetFPGA SUME Virtex-7 FPGA boards and eight Dell T720 servers.