MINNIE: an SDN World with Few Compressed Forwarding Rules

Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, Joanna Moulierac, Dino Lopez Pacheco, Guillaume Urvoy-Keller

To cite this version:

Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, Joanna Moulierac, et al.. MINNIE: an SDN World with Few Compressed Forwarding Rules. [Research Report] RR-8848, INRIA Sophia-Antipolis; I3S. 2016. hal-01264387

HAL Id: hal-01264387
https://inria.hal.science/hal-01264387

Submitted on 29 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MINNIE: an SDN World with Few Compressed Forwarding Rules

Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frederic Giroire, Joanna Moulierac, Dino Lopez Pacheco, Guillaume Urvoy-Keller

RESEARCH REPORT
N° 8848
Janvier 2016
Project-Team COATI

ISSN 0249-6399 ISRN INRIA/RR--8848--FR+ENG

MINNIE: an SDN World with Few Compressed Forwarding Rules

Myriana Rifai*, Nicolas Huin*†, Christelle Caillouet*†, Frederic Giroire*†, Joanna Moulierac*†, Dino Lopez Pacheco*, Guillaume Urvoy-Keller*

Project-Team COATI

Research Report n° 8848 — Janvier 2016 — 33 pages

Abstract: Software Defined Networking (SDN) is gaining momentum with the support of major manufacturers. While it brings flexibility in the management of flows within the data center fabric, this flexibility comes at the cost of smaller routing table capacities. Indeed, the Ternary Content Addressable Memory (TCAM) needed by SDN devices has smaller capacities than CAMs used in legacy hardware.
In this paper, we investigate compression techniques to maximize the utility of SDN switches forwarding tables. We validate our algorithm, called MINNIE, with intensive simulations for well-known data center topologies, to study its efficiency and compression ratio for a large number of forwarding rules. Our results indicate that MINNIE scales well, being able to deal with around a million of different flows with less than 1000 forwarding entry per SDN switch, requiring negligible computation time.
To assess the operational viability of MINNIE in real networks, we deployed a testbed able to emulate a $k = 4$ fat-tree data center topology. We demonstrate on one hand, that even with a small number of clients, the limit in terms of number of rules is reached if no compression is performed, increasing the delay of new incoming flows. MINNIE, on the other hand, reduces drastically the number of rules that need to be stored, with no packet losses, nor detectable extra delays if routing lookups are done in ASICs.
Hence, both simulations and experimental results suggest that MINNIE can be safely deployed in real networks, providing compression ratios between 70% and 99%.

Key-words: Software Defined Networks, data center networks, routing tables, compression, TCAM memory

* University of Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
† Inria Sophia Antipolis

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

MINNIE: un Monde SDN avec Peu de Règles de Forwarding

Résumé : Les Software Defined Networks (SDN) prennent de l'ampleur grâce à l'appui de grands fabricants. Bien qu'il apporte de la souplesse dans la gestion des flots au sein des centre de données, cette flexibilité se fait au détriment des capacités de table de routage, qui sont plus petites. En effet, la mémoire requis par les périphériques SDN (TCAM) a une capcité réduite par rapport à celle utilisés dans le matériel existant.

Dans cet article, nous étudions des techniques de compression pour maximiser l'utilisation des tables de routage SDN. Nous validons notre algorithme, appelé MINNIE, avec des simulations intensives sue des topologies de centre de données connues, afin d'étudier son efficacité et le taux de compression pour un grand nombre de règles. Nos résultats indiquent que MINNIE passe bien à l'échelle, être en mesure de traiter environ un million de différents flux avec moins de 1000 entrée de transfert par commutateur SDN, nécessitant un temps de calcul négligeable.

Pour évaluer la viabilité de MINNIE dans les réseaux réels, nous déployons une plateforme capable d'émuler un $k = 4$ fat-tree. Nous démontrons d'une part, que même avec un petit nombre de clients, la limite en termes de nombre de règles est atteinte si aucune compression est réalisée, augmentant le délai des nouveaux flots arrivants. MINNIE, d'autre part, réduit considérablement le nombre de règles qui doivent être stockées, sans pertes de paquets, ni délai supplémentaires visibles. Par conséquent, les résultats de simulations et expérimentaux suggèrent que MINNIE peut être déployé en toute sécurité dans des réseaux réels, offrant des taux de compression entre 70 % et 99 %.

Mots-clés : Software Defined Networks, réseaux programmables, réseaux de centre de données, table de routage, mémoire TCAM

1 Introduction

In classical networks, routers compute routes using distributed routing protocols such as OSPF (Open Shortest Path First) [22] to decide on which interfaces packets should be forwarded. In Software Defined Networks (SDN), one or several controllers take care of route computations and routers become simple forwarding devices. When a packet arrives with a new destination for which no routing rule exists, the router[1] contacts a controller that provides a route to the destination. Then, the router stores this route as a rule in its SDN table and uses it for next incoming matching packets. This separation of the control plane from the data plane allows a smoother control over routing and an easier management of the routers.

Also, SDN networks aim at applying flow-based forwarding rules instead of destination-based rules (as in legacy routers) to provide a finer control of the network traffic. For instance, in OpenFlow 1.0 [2], forwarding decisions can be made taking into account from zero up to a maximum of 12 fields of a TCP or UDP packet. When any of the 12 fields should be ignored when forwarding a packet, such a field is set to “don’t care bits”. Due to the complexity of SDN forwarding rules, SDN forwarding devices need Ternary Content-Addressable Memories (TCAMs) to store their routing table (as classical CAM can only perform binary operation)s. However, TCAMs are more power hungry, expensive and physically bigger than binary CAMs available in legacy routers. Consequently, the available TCAM memory in routers is limited. Indeed, a typical switch supports in the order of a thousand 12-tuple flows; the actual number ranges from 750 to 4000 [1].

Undoubtedly, emerging switches will support larger rule tables [6], but TCAMs still introduce a fundamental trade-off between rule-table size and other concerns like cost and power. The maximum size of routing tables is thus limited and represents an important concern for the deployment of SDN technologies. This problem has been addressed in previous works, as discussed in Section 2, using different strategies, such as routing table compression [11, 15], or distribution of forwarding rules [8].

In this work, we examine a more general framework in which *table compression* using wildcard rules is possible. Compression of SDN rules was discussed in [11]. The authors propose algorithms to reduce the size of the tables, but only by using a default rule. We consider here a stronger compression methodology in which any packet header field may be compressed. Considering *multiple field aggregation* is an important improvement as it allows a more efficient compression of routing tables, leaving more space in the TCAM to apply advanced routing policies, like load-balancing and/or to implement quality of service policies. In the following, we focus on compression of rules based on sources and destinations. However, our solution also applies if other fields are considered such as ToS (Type of Service) field or transport protocol.

In this paper, we tackle the problem of dynamically routing traffic demands inside a data center network using SDN technologies. Our contributions are the following:

- We provide an algorithm, MINNIE, in Section 3, which routes the traffic and compress routing tables to satisfy link capacity and routing table size constraints of the different forwarding devices. The compression can be done on different flow fields allowing advanced routing policies.
- The routing is done dynamically, meaning that the routing and compression decisions are taken online when a new flow arrives. We show that compressing the tables at the right moment can lead to significant gains in Section 5.
- We first validate the algorithm by *extensive simulations* on several *well-known data center topologies* described in Section 4. We show it can be used in *large environments* in Section 5: it scales

[1] In the following, we make no distinction between routers/switches, packets/frames and routing/forwarding tables using these terms in their general sense.

[2] `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf`

well and can deal with around a *million of different flows* with less than 1000 entries in the routing tables and with negligible compression time.

- We then validate our simulations with a *testbed composed of SDN hardware*, described in Section 6. We study different metrics, namely the *delay* introduced by the communications with the controller, the potential increase of *loss rate* due to handling of dynamic routing and the *load of the controller* with and without compression.

- Our results (Sections 7, 8 and 9) show that we are able to *minimize the number of entries in the switches, while successfully handling client's dynamics and maintaining the network stability.*

- Section 10 aims at summarizing the results obtained via simulation and within our testbed, in order to *pick operational parameters* that would fit any traffic and topology scenario.

2 Related work

To support a vast range of network applications, SDN has been designed to apply flow-based rules, which are more complex than destination-based rules in traditional IP routers. As explained in the previous section, the complexity of the forwarding rules are well supported by TCAMs. However, as TCAM is expensive and extremely power-hungry, the on-chip TCAM size is typically limited.

Many existing studies in the literature have tried to address this limited rule space problem. For instance, the authors in [18] and [4] try to compact the rules by reducing the number of bits describing a flow within the switch by inserting a small tag in the packet header. This solution is complementary to ours, however, it requires a change in: (i) packet headers and (ii) in the way the SDN tables are populated. Also, adding an identifier to each incoming packet is hard to be done in the ASICs since this is not a standard operation, causing the packets to be processed by the CPU (a.k.a. the slow-path), strongly penalizing the performance of a forwarding device and the traffic rate. Another approach is to compress policies on a single switch. For example, the authors in [3, 20, 21] have proposed algorithms to reduce the number of rules required to realize policies on a single switch.

Several works have proposed solutions to distribute forwarding policies while managing rule-space constraints at each switch: [8, 17, 16, 23]. However, no compression mechanisms are proposed in these works. For example, in [23], the authors propose OFFICER. It creates a default path for all the communications, and later, some deviations are introduced from this path using different policies to reach the destination. According to the authors, the Edge First (EF) strategy, where the deviation is placed to minimize the number of hops between the default path and the target one, offers the best trade-off between the required QoS and forwarding table size. Note however, that applying this algorithm could unnecessarily penalize the QoS of flows when the switches' forwarding tables are rarely full.

To the best of our knowledge, the closest papers to our work are from [15, 7, 11]. In [15] the authors introduce XPath which identifies end-to-end paths using path ID and then compresses all the rules and pre-install the necessary rules into the TCAM memory. We compare our results with the one of XPath in Section 5.4. MINNIE uses fewer rules even in the case of an all-to-all traffic as XPath codes the routes for all shortest paths between sources and destinations. This is at the cost of less path redundancy which is useful for load-balancing and fault tolerance. Network operators should consider this trade-off when choosing which method to use. Note that MINNIE is even more efficient when the traffic is far from all-to-all and when only few shortest paths are used between a source and a destination. In [7] the authors suggest SDN rule compression by following the concept of longest prefix matching with priorities using the Espresso [27] heuristic and show that their algorithm leads to 17% savings only. We succeed in reaching better compression ratios using MINNIE. Last, [11] addresses the problem of compressing routing tables using default rule only in case of Energy-Aware Routing. We extend this solution by considering other types of compression.

Flow	Output port
(0, 4)	Port-4
(0, 5)	Port-5
(0, 6)	Port-5
(1, 4)	Port-6
(1, 5)	Port-4
(1, 6)	Port-6
(2, 4)	Port-4
(2, 5)	Port-5
(2, 6)	Port-6

(a) Without Compression

Flow	Output port
$(0, 4)$	Port-4
$(1, 5)$	Port-4
$(2, 4)$	Port-4
$(2, 5)$	Port-5
$(0, *)$	Port-5
$(*, *)$	Port-6

(b) MINNIE: Source table

Flow	Output port
(1, 4)	Port-6
$(1, 5)$	Port-4
$(0, 6)$	Port-5
$(*, 4)$	Port-4
$(*, 5)$	Port-5
$(*, *)$	Port-6

(c) MINNIE: Destination table

Flow	Output port
(0, 5)	Port-5
(0, 6)	Port-5
(1, 4)	Port-6
(1, 6)	Port-6
(2, 5)	Port-5
(2, 6)	Port-6
$(*, *)$	Port-4

(d) MINNIE: Default only

Flow	Output port
(1, 5)	Port-4
(2, 6)	Port-6
$(1, *)$	Port-6
$(*, 4)$	Port-4
$(*, *)$	Port-5

(e) Optimal solution (ILP)

Table 1: Examples of routing tables: (a) without compression, (b) compression by the source, (c) compression by the destination, (d) default rule only, and (e) routing table with minimum number of rules given by Integer Linear Program.

In this work, we study a new and original way to compress the rules in SDN tables using aggregation by source or destination. Previously, in [25] which is the short version of this document, we have already introduced the idea of aggregating the rules by source or destination. In this document, we present a deep evaluation of our solution by performing extensive simulations on several different data center architectures, and by studying the benefits of MINNIE in large environments to prove the scalability of our method. We also extend our previous work by proposing compression in all switches thanks to the introduction of level-0 switches at the servers, as we will explain in Section 6.2. Indeed, in [25], we did no compression of the access switch routing tables. This article also presents experimental results obtained with an optimal configuration of the hardware switch (i.e., execution of TCAM operations only) to detect any negative impact of MINNIE.

3 Modeling of the problem and Description of MINNIE algorithm

We represent the network as a directed graph $G = (V, A)$. A vertex is a router and an arc represents a link between two routers. Each link has a maximum capacity and the number of rules of a router is limited by the size of its routing table. For a set of demands $\mathcal{D}$, a routing solution consists in assigning to each demand a path in a way that the capacity constraints and the table size constraints are respected.

We define a routing rule as a triplet (s, t, p) where s is the source of the flow, t its destination and p the outgoing port of the router for this flow. To aggregate the different rules, we use *wildcard rules* that can merge rules by source (i.e. $(s, *, p)$), by destination (i.e. $(*, t, p)$) or both (i.e. $(*, *, p)$, the default rule). Table 1 shows an example of a routing table and its compressed version using different strategies. Table 1(a) gives the routing table without compression, Table 1(d) the table using default port compression and Table 1(e) the minimal routing table using a mix of compressions by sources and by destinations.

Note that rules have priorities over one another (based on their ordering) in case multiple rules correspond to a flow. For example, in the solution with the minimum number of rules (Table 1(e)), rule $(1, *, 6)$ must have a higher priority than (and so placed before) rule $(*, 4, 4)$ in the table, otherwise the flow $(1, 4)$ would be routed through Port-4, which is not the routing decision taken in Table 1(a).

3.1 MINNIE: Compression phase

Since even the compression of a single table described in the previous section is NP-Hard [10], we propose the following efficient heuristic: it first computes three compressed routing tables (aggregation by source, by destination and by the default rule) and then chooses the smallest one, as explained in more details below. The heuristic is efficient both in practice, as our results show that it leads to high compression ratios, and in theory, as it has been shown that it provides a 3-approximation of the compression problem [10].

Given a routing table such as the one given in Table 1(a), the algorithm first considers all the sources one by one. For each source s, we find the most occurring port p^*, and replace all the matching rules with $(s, *, p^*)$ (Table 1(b)). The remaining rules $(s, t, p \neq p^*)$ stay unchanged and have priority over the source wildcard rule. Once all the sources have been considered, we do a pass over all the wildcard rules. We aggregate them using the most occurring port that becomes the default port. The default port rule has the lowest priority of all the rules. For the second compressed routing table (Table 1(c)), we do the same compression considering the aggregation by destination with $(*, t, p^*)$ rules. As for the third table (Table 1(d)) a single aggregation using the best default port is performed, i.e., the most occurring port in the routing table becomes the default port. We then choose the smallest routing table of the three.

3.2 MINNIE: Routing phase

We propose an efficient routing heuristic which spreads flows over the network to avoid overloading a link or a table using a shortest-path algorithm with an adaptive metric.

We route the demands one by one. For every demand between source s and destination t with a load d, we first build a weighted digraph (G_{st}, w) representing the residual network:

- G_{st} is a subgraph of G where an arc (u, v) is removed if its capacity is less than d or if the flow table of the router u is full and does not contain any wildcard rule for (s, t, p_v) (where p_v represents the output port of u towards v). Note that, when a table is full and compressed, a node u has only one outgoing arc (to the node v), corresponding to the first existing rule of the form $(s, *, p_v)$, $(*, t, p_v)$ or to the default rule $(*, *, p_v)$. As more tables get full, the number of nodes with only one outgoing arc increases, reducing the size of the graph.

- The weight w_{uv} of a link depends on the overall flow load on the link and the table's usage of router u. We note w^c_{uv} the weight corresponding to the link capacity and w^r_{uv} the weight corresponding to the rule capacity. They are defined as follows:

  $$w^c_{uv} = \frac{\mathcal{F}_{uv} + d}{C_{uv}}$$

  where C_{uv} is the capacity of the link (u, v) and $\mathcal{F}_{uv}$ the total flow load on (u, v). The more the link is used, the heavier the weight is, which favors the use of lower loaded links allowing load-balancing. And

  $$w^r_{uv} = \begin{cases} \frac{|R_u|}{S_u} & \text{if } \nexists \text{ wildcard rule for } (s, t, v) \\ 0 & \text{otherwise} \end{cases}$$

  where S_u is the maximum table size of router u and R_u is the set of rules for router u. The weight is proportional to the usage of the table.

  The weight w_{uv} of a link (u, v) is given by:

  $$w_{uv} = 1 + 0.5 * w^c_{uv} + 0.5 * w^r_{uv}$$

When (G_{st}, w) is built, we compute a route for the demand by finding a shortest path between s and t in the digraph minimizing the weight w. Once a shortest path is found, for each arc (u, v) of the path, we add the rule (s, t, v) to the router u, if no corresponding wildcard rule exists, and the capacity of the arc is decreased by the load of the demand d. If the table is full or has reached a given threshold, we **compress** it using the algorithm described previously. If no path is found (which can occur when the links are overloaded) the demand is ignored, leading to packet drops.

4 Simulation of MINNIE on different data center topologies

In this section, we present the different scenarios (and performance metrics) that we consider to study the behavior of MINNIE for a wide variety of data center architectures.

4.1 Simulation settings

We present in this section the different scenarios studied via simulations, the traffic patterns and metrics that will be evaluated. All simulations were carried out on a computer equipped with a 3.2GHz 8 Core Intel Xeon CPU and 64 GB of RAM.

4.1.1 Scenarios

We ran simulations under three different scenarios:

- **Scenario 1: No compression**. We only use the routing module of MINNIE and fill up the routing tables without compressing them. This scenario serves as a baseline for measuring the efficiency of MINNIE.
- **Scenario 2: Compression at the end of the simulation**. We compress the routing tables of every switch once at the end of the simulation, when all the forwarding rules have been stored assuming an unlimited capacity of the routing table. We use it to test deterministically the compression module of MINNIE.
- **Scenario 3: MINNIE (Dynamic compression at a fixed threshold)**. We validate MINNIE with a threshold of 1000 rules, which represents the routing table limit. This scenario aims at testing MINNIE in a scenario closer to real life. The capacity of 1000 rules has been chosen as it corresponds to the number of entries supported by the TCAM memory of typical switches [1].

4.1.2 Traffic patterns

For all scenarios, we consider an all-to-all traffic in which every single server establishes a connexion to all other servers. We consider this situation to test MINNIE in the most extreme scenario in terms of number of flows, and thus, in terms of number of rules. For each topology, we introduce the flows randomly one by one. For each new flow, the algorithm selects the best route considering the adaptive metric and then populates the routing tables along the route with the new rules.

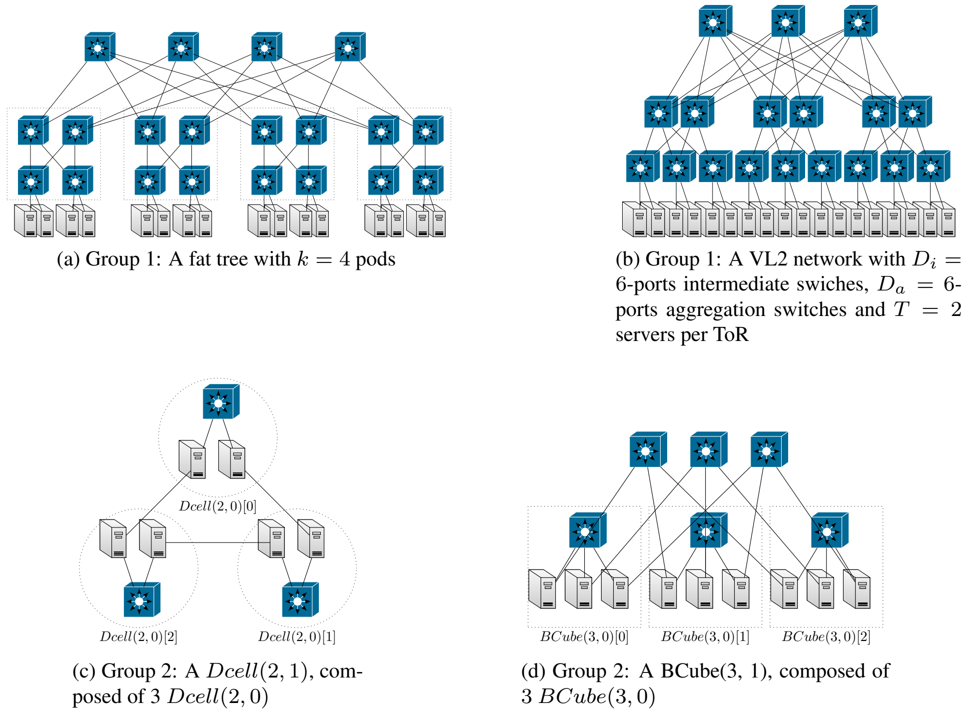
4.1.3 Metrics

To assess the efficiency of MINNIE, we measure the following metrics:

- Average compression ratio: *compression ratio* = $1 - \frac{\textit{number of rules of a switch}}{\textit{number of flows passing through the switch}}$.
- Number of compressions performed by a switch during the simulation.

- Number of flows passing through a switch (maximum and average over all switches).
- Number of rules, total and per switch (maximum and average over all switches).
- Computation time for compressing a table and for routing a flow.
- Maximum number of servers which can be installed on a data center topology without going beyond a forwarding table size of 1000 rules.

4.2 Data Center Architectures



(a) Group 1: A fat tree with $k = 4$ pods

(b) Group 1: A VL2 network with D_i = 6-ports intermediate swiches, D_a = 6-ports aggregation switches and T = 2 servers per ToR

(c) Group 2: A $Dcell(2, 1)$, composed of 3 $Dcell(2, 0)$

(d) Group 2: A BCube(3, 1), composed of 3 $BCube(3, 0)$

Figure 1: Example of topologies studied.

To test the efficiency of MINNIE, we considered the most common data center architectures: Fat Tree[2], VL2 [12], BCube [13] and D-Cell [14]. For each family of architecture, we considered topologies of different sizes hosting from few units to about 3000 end points. These end points can be either servers or IP subnets, grouping thousands of different machines. In the following, for simplicity, we often use the term *server* for both cases. The number of flows routed in the topologies can thus reach few millions.

The architectures considered during these simulations can be classified into two different groups:

- **Group 1**, in which servers only act as end hosts includes Fat Tree and VL2.
- **Group 2**, in which servers also act as forwarding devices (similarly to switches) includes BCube and D-Cell.

We detail below how we chose the different set of parameters (number of switches, level of recursion...) to build these topologies.

Fat-Tree. The fat tree is one of the most well-known architectures. The switches are divided into three categories: core, aggregation and access (or ToR for Top of the Rack) switches. A k-fat tree is composed of k pods of k switches and $k^2/4$ core switches. Every switch possesses k ports. Inside a pod, aggregation and edge switches form a complete bipartite graph. Each core switch is connected to every pod via one of the $k/2$ aggregation switches. Every ToR switch has a rack composed of $k/2$ servers. A 4-fat tree is shown as example in Figure 1a.

For our simulations, to build fat trees with up to 3000 servers, we considered k-fat trees for k between 4 and 22.

VL2. The VL2 architecture is also composed of three layers of switches: intermediate, aggregation and ToR switches. The intermediate and aggregation switches are connected together to form a complete bipartite graph. Each ToR is connected to two different aggregation switches. Three parameters control the number of switches of each layer and the number of servers of the architecture: D_a represents the number of ports of an aggregation switch, D_i the number of ports of an intermediate switch and T the number of servers in the rack of a ToR switch. Figure 1b shows a $VL2(D_a = 6, D_i = 6, T = 2)$. The topology has $D_a/2$ (3 in the example) aggregation switches, D_i (6 in the example) intermediate switches, $D_aD_i/4$ (9 in the example)ToR switches and $TD_aD_i/4$ (18 in the example) servers.

For our simulations, we chose the parameters of the topologies to ensure that every switch has the same number of ports, that is VL2$(2k, 2k, 2k-2)$ for k between 2 and 11.

Dcell. The Dcell architecture is a topology in which both servers and switches act as forwarding devices. The topology is built recursively. The basic block is the level-0 Dcell, $Dcell(n, 0)$, where n servers are connected to a unique switch. From a $Dcell(n, l-1)$, composed of $s(n, l-1)$ servers, a $Dcell(n, l)$ can be build by connecting each server of a $Dcell(n, l-1)$ to a different $Dcell(n, l-1)$. This builds a $Dcell(n, l)$ containing $(s(n, l)+1) \times Dcell(n, l-1)$. For example, a $Dcell(2, 0)$ is composed of 2 servers ($s(n, 0) = n$) and to create a $Dcell(2, 1)$ as shown in Figure 1c, ($s(2, 0)+1 = 3$) $Dcell(2, 0)$ are interconnected.

In our simulations, we compare topologies with one level of recursion (referenced as $Dcell(l = 1)$), with n between 1 and 54, and topologies with two levels of recursion (referenced as $Dcell(l = 2)$), with n between 1 and 7.

BCube. BCube is the second architecture in which the servers also act as forwarding devices. Again, it is a recursive construction. The building block is a $BCube(n, 0)$, composed of n servers connected to a single switch. The level l being composed from multiple $l-1$ levels. Unlike in the construction of Dcell, in which the recursion connect servers together, the construction of BCube is done by connecting the servers via new switches. The number of switches added to make a BCube of level l is equal to the number of servers in a BCube of level $l-1$. Each switch is then connected to one server of every BCube of level $l-1$ and each servers to $l+1$ switches – see the BCube$(3, 1)$ in Figure 1d.

Like for Dcell topologies, the same number of servers can be obtained with different levels of recursion. We consider levels of recursion up to level 3.

5 Efficiency of MINNIE through simulations

In this section, we validate MINNIE through simulations over the set of topologies described in Section 4.1.1. We demonstrate in this section that MINNIE works well for different topologies and different sizes of data centers. We first analyze the compression rates that can be obtained by compressing large tables. Then, we show that if tables are compressed all along the simulation as soon as the limit is reached, then the compression module is much more efficient and the compression ratio reaches 90% for

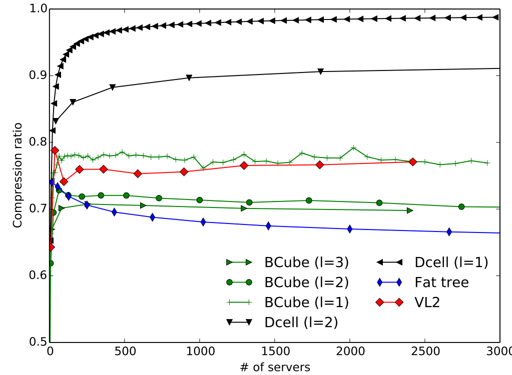


Figure 2: Compression ratio for the different topologies in Scenario 2.

some topologies. We then investigate the efficiency of MINNIE when considering around 1000 servers in multiple topologies. We show the efficiency of our method by comparing the results of MINNIE with XPath [15]. Finally, we present the routing and compression time of these different topologies.

For each family of topologies, we present the results for the three scenarios described in Section 4.1.1, referenced respectively as *No compression*, *Compression at the end* and MINNIE.

5.1 Efficiency of the Compression module

Efficiency of the Compression module in Scenario 2.

The efficiency of the *compression module* of MINNIE can be observed in Figure 2 where we compare the compression ratios between *No compression* and *Compression at the end*. In this figure we observe that Dcell, BCube and VL2 topologies follow a similar phenomenon. They all feature a sharp increase of the compression ratio when the number of servers is between 0 and 100: for example, the ratio raises from 0.62 to 0.84 for Dcell(l=2). Then, for larger number of servers, the compression ratio levels off. On the other hand, fat tree topologies have a different behavior and do not experience the increase phase and the curve is almost flat all along the simulation.

In the flat phase, compression ratios are between 60% and 80% for the three families BCUBE, VL2 and Fat tree, and even reach values between 85% and 99.9 % for Dcell. **In summary, the compression module of MINNIE can attain a minimum of 60% savings in memory.**

Compression frequency. In Figure 3 we observe the total number of compressions executed for the different topologies. Group 1 topologies reach a maximum of 516 compressions for the 18 fat tree (and 301 for VL2(20, 20, 18)). This represents an average of about 1 compression per switch for the fat tree topology and less than 6 compression for VL2. However, Group 2 shows a higher number of compressions, with a maximum of almost 6000 compressions for a $BCube(53, 1)$ (in average, 54 compressions per forwarding device). This difference is due to the near saturation of most of the switches in Group 2 topologies. In these nearly saturated tables, the compression leaves a table that is close to the 1000 limit and thus, the table is compressed only after a few new flows are added.

5.2 Efficiency of the routing and compression modules

MINNIE is composed of a routing and a compression module. When the number of rules reaches the 1000 limit, MINNIE triggers the compression module. This dynamic behavior allows to efficiently route

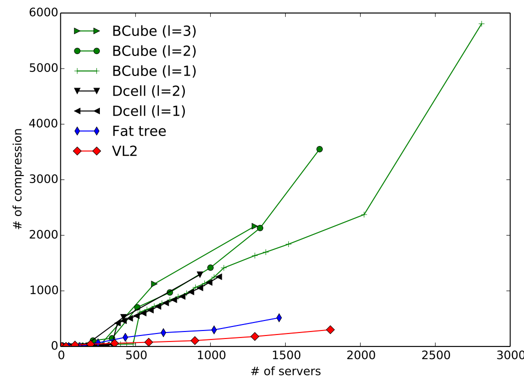


Figure 3: Number of compression executed for different topologies

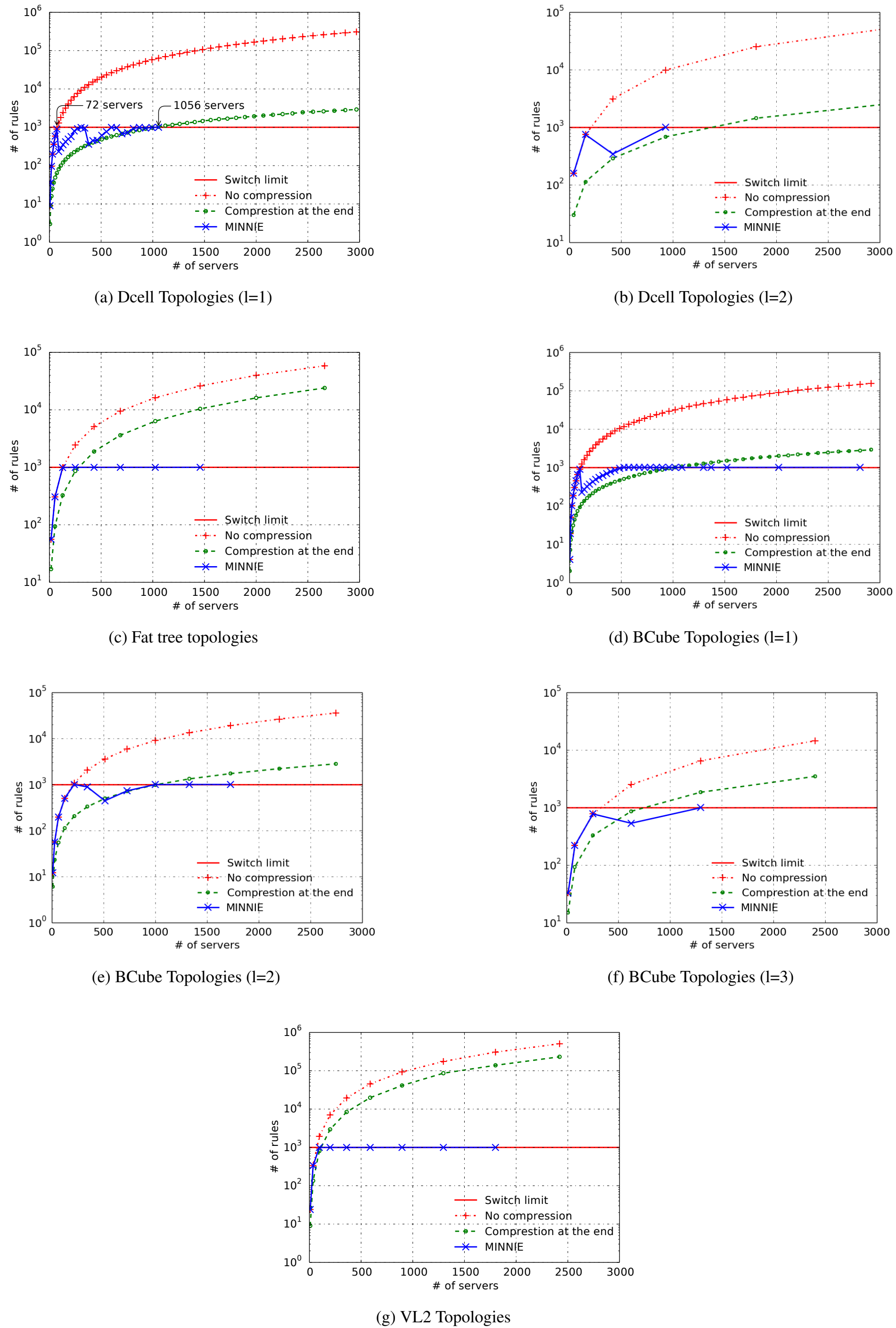
traffic *without overloading the routing tables* on topologies where the number of servers increases. Figure 4 presents the maximum number of rules on a device (a router or a server depending on the family of topology) as a function of the number of servers for the different families of topologies. We remark that the curve for MINNIE first follows the *No compression* one until reaching the 1000 limit. Indeed, during this first phase, MINNIE performs no compression at all as the limit is not attained. Then, MINNIE triggers compression regularly and manges to keep all routers' table below the limit of 1000. When performing compression, MINNIE has introduced wildcard rules in the routing tables, and the new incoming flows will follow these paths in priority. Therefore, MINNIE deals with the same number of flows as *No Compression* with less than 1000 entries while *No Compression* needs between 10^4 and 10^6 entries. Note that some points for MINNIE are not depicted. Indeed, in Figure 4, we present only the results in which all the flows are routed without overloading the routing tables. As soon as one request cannot be routed and when the routing tables cannot be further compressed, the simulations are stopped.

This phenomenon can be clearly seen for Dcell(l=1) topologies in Figures 4a. Without compression, only 72 servers can be deployed in a $Dcell(8, 1)$ without overloading tables while MINNIE allows to deploy 1056 servers with a $Dcell(32, 1)$. This represents a 15 fold increase compared to *No compression*. The number of servers which can be deployed with Dcell topologies having two levels of recursion (Figure 4b) is similar: 930 with a $Dcell(5, 2)$ when running MINNIE and less than 200 with No compression.

Another key observation is that MINNIE can reach or even outperforms *Compression at the end* without exceeding the limit of number of rules. Indeed, if we consider for example Fat tree topologies in Figure 4c, without compression, the largest fat tree which can be deployed with a rule limit of 1000 is an 8-fat tree with 128 servers and 992 rules. With compression at the end, the number of servers which can be deployed would be around 256. However, we see that MINNIE succeeds in deploying an 18-fat tree with 1458 servers without having overloading issues. This is a 6 fold increase compared to *Compression at the end. This is due to the fact that by compressing online, i.e., when flows are introduced,* MINNIE *impacts the routing of the following flows*. Because of the metric used in the routing module, the algorithm will prefer to select shortest paths using wildcards as they do not increase the number of rules. This allows to obtain better compression ratios.

The phenomena appears also for BCube topologies (Figures 4d, 4e, 4f) and with a striking intensity for VL2 topologies (Figure 4g). When compressing at the end, up to 96 servers can be deployed without reaching the table size limit (and only 36 without compression). With MINNIE, this number can be pushed up to 1800 servers. This is an **impressive** 36 fold increase!

Difference of behavior inside a family of topologies. We notice in Figure 2 and 4 a difference of



(a) Dcell Topologies (l=1)

(b) Dcell Topologies (l=2)

(c) Fat tree topologies

(d) BCube Topologies (l=1)

(e) BCube Topologies (l=2)

(f) BCube Topologies (l=3)

(g) VL2 Topologies

Figure 4: Maximum number of rules on a forwarding device as a function of the number of servers for different data center architectures.

Topology	servers #	switches #	links #	Avg ports #	Flow #			Rule w/ comp #			Average Comp. Ratio	Computation time in average (ms)	
					Total	per switch Max	per switch Average	Total	Max	Average		Paths	Comp.
Group 1													
4-Fat tree (64)	1024	20	1056	54.4	917 504	454 244	216 268	8923	999	446	∼ 99.60	0.17	13
8-Fat tree (8)	1024	80	1280	19.2	1 015 808	649 044	61 030	25 853	999	323	∼ 99.61	0.21	7
16-Fat tree (1)	1024	320	3072	16	1 040 384	630 998	15 897	97 173	999	303	∼ 98.42	0.30	5
VL2(16, 16, 14)	896	88	384	16	790 272	261 266	42 906	59 237	1000	673	∼ 97.90	0.15	4
VL2(8, 8, 64)	1024	28	612	∼ 41.1	983 040	423 752	161 499	22 394	1000	799	∼ 99.45	0.19	11
VL2(16, 16, 16)	1024	88	1152	∼ 17.5	1 032 192	276 575	56 040	57 078	1000	648	∼ 98.39	0.18	4
Group 2													
Dcell(32, 1)	1056	33	1584	∼ 2.91	1 114 080	63 787	4893	123 655	1000	113	∼ 97.23	0.09	2
Dcell(5, 2)	930	186	1860	∼ 3.33	863 970	11 995	5716	717 018	994	642	∼ 87.84	0.19	2
BCube(32, 1)	1024	64	2048	∼ 3.77	1 047 552	37 738	3734	358 204	999	329	∼ 86.04	0.19	2
BCube(10, 2)	1000	300	3000	∼ 4.62	999 000	10 683	4153	849 316	998	653	∼ 80.85	0.25	2
BCube(6, 3)	1296	864	5184	4.8	1 678 320	7852	5184	1 795 400	991	831	∼ 83.18	0.49	4

Table 2: Comparison of the behavior of MINNIE for different families of topologies with around 1000 servers each. For the fat tree topologies, we tweak the number of clients per server to obtain 1024 ”servers”.

behavior inside a family of topologies. For a given family of data centers, different topologies can host a similar number of servers. For example, Dcell(32,1) and Dcell(5,2) host around 1000 servers, as well as BCube(32,1), BCube(10,2) and BCube(6,3). But the behavior of these topologies is sensibly different: for example, the average number of rules is 113 for a Dcell(32,1) compared to 642 for a Dcell(5,2). We see that the compression ratio of the family Dcell(l=1) is higher (more than 95% when the number of servers is greater than 200) than the one of Dcell(l=2) (more than 85% when the number of servers is greater than 200). **Hence, the choice of the best set of parameters for a given family of topologies is very important.** In order to answer this question, we study in the following section all these topologies with similar number of servers (around 1000).

5.3 Comparison of MINNIE effect on topologies with 1000 servers

Table 2 sums up the effect of MINNIE on the different topologies with a similar number of servers (around 1000), hence a similar number of flows to route. We detail below the different parts of the table, highlighting the key conclusions to draw.

Topology characteristics. The first part of the table provides basic information about the topologies. **Even with a similar number of servers, the topologies are very different** in terms of number of switches (between 20 and 903), links (between 1056 and 5184) and average number of ports per switch (between 2.9 and 54.4).

Flows in the network. The second part of the table reports the number of flows introduced in the network during the simulation. These topologies behave very differently in terms of number of flows per device: the average number of rules ranges from 3734 to 216 000 and the maximum number of rules ranges from 7800 to 650 000. Two explanations can be given for these differences. First, the topologies have very different numbers of switches (from 20 to 864). Secondly, in the topologies of Group 2, servers also act as switches, and thus also host some rules, leading to a lower average number per device.

Compressing with MINNIE. The third part of the table represents the effect of using MINNIE on the: number of rules, average compression ratio and computation time. **MINNIE succeeds to route the traffic on all the topologies without exceeding the limit of** 1000 **rules per device** (maximum number of rules between 989 and 1000). We also observe that with 1000 servers **MINNIE allows to attain an average compression ratio higher than 80%**. As for the computation time we notice that **MINNIE dynamically computes the route with a sub-millisecond delays** as the maximum routing computation time is 0.49ms for BCube(6,3). And finally, we can observe that **compressing the rules with MINNIE will cost less than 13ms delay in all topologies**.

DCNs	Server to Server	
	XPath	MINNIE
BCube(4, 2)	108	56
BCube(8, 2)	522	443

(a) Comparison with MINNIE for paths between servers

DCNs	ToR to ToR		Server to Server
	XPath	MINNIE	MINNIE
8-Fat tree	116	27	272
16-Fat tree	968	116	6351
32-Fat tree	7952	482	113 040
64-Fat tree	64 544	1925	-
VL2(20, 8, 40)	310	135	138 354
VL2(40, 16, 60)	2820	1252	-
VL2(80, 64, 80)	49 640	22 957	-

(b) Comparison with MINNIE: for paths between servers and paths between level 1 switches

Table 3: Comparison of maximum number of rules on a switch between XPath and MINNIE (between servers or ToRs).

5.4 Comparison with XPath

We compare MINNIE with another compression method of the literature, XPath [15]. XPath combines re-labeling and aggregation of paths. Each path is assigned to an ID. Two paths can share the same ID if they are either convergent or disjoint but not if they are divergent. The assignment of IDs is then based on a prefix aggregation. This method requires that, for every request in the data center, an application must contact the controller to acquire the corresponding ID of the path to its destination.

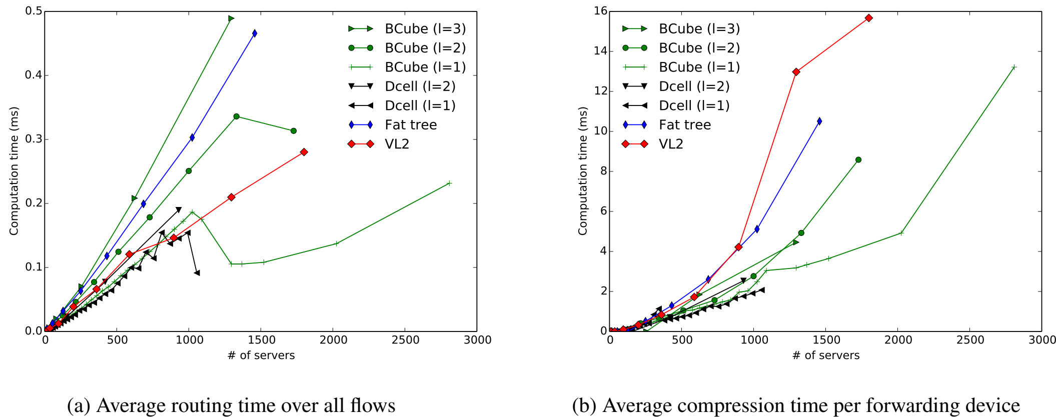
In Table 3, we compare the maximum number of rules installed on a forwarding device between XPath and MINNIE. In MINNIE, we consider all the demands between servers even if they act only as end hosts but in XPath, only the path between ToRs are considered for the standard architecture (VL2, Fat tree). So for an accurate comparison, we apply the same principle to MINNIE by only considering demands between ToRs. Since they also consider bigger table size of 144 000 entries, the limit is set to 144 000 for MINNIE too. MINNIE requires a lower number of rules to be installed than XPath on every architecture while both dealing with all possible (source,destination) flows. This can be explained by the fact that XPath installs rules for all possible paths for every source/destination pair before compressing while MINNIE only considers one path per demand.

5.5 Execution time of MINNIE

Finally, we study the execution time of MINNIE in order to assess if it is a viable solution in practice.

Routing time. When a new flow arrives, the controller has to compute its path in the network and the set of rules to be installed in the switches along the path. We plot in Figure 5a the average time for this operation. Recall that, to compute the paths we used the Dijkstra algorithm with the metric w_{uv} and residual graph G_{st} described in Section 3. The longest average time is about 0.42 ms which corresponds to the 18-fat tree (with 1458 servers and 405 switches), whereas the shortest routing time happens for VL2, Dcell(l=1) and BCube(l=1) which have a small number of shortest paths between two routers. On the contrary, the fat tree and BCube(l=3) experience a longer routing time explained by the large number of possible paths between two servers. Note that even if fat tree and VL2 have a similar shape, the latter topology has significantly fewer switches and edges, which explain the smaller number of possible paths and therefore the smaller routing time. Nevertheless, **for all of the studied topologies, the routing time is small and we will see in Section 8 that the delays of the packets are not significantly impacted.**

Moreover, we observe a surprising behavior for some topologies. In most cases, the computation time is globally increasing with the size of the topologies. However, Dcell(l=1), BCube(l=1), and BCube(l=2) experience a drop in computation time: For example, the computation time for BCube(l=1) topologies increases to 0.18ms for 1024 servers, then drops to 0.10 ms for 1350 servers to increase again to 0.22



(a) Average routing time over all flows

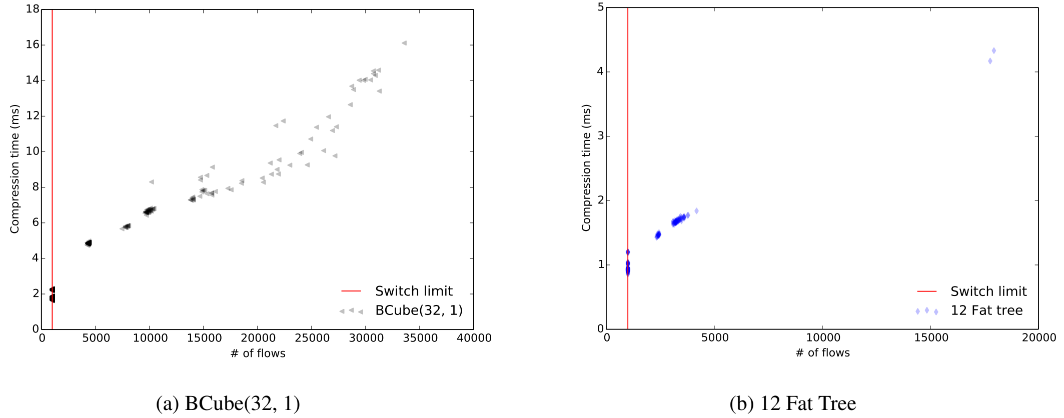
(b) Average compression time per forwarding device

Figure 5: Computation time for the compression and routing phases for different topologies.

ms for 2800 servers. This behavior is caused by the saturation of a large number of switches of the topology when the number of flows becomes high during the simulation. A switch is *saturated* when the compression module can no longer reduce the size of the table below the 1000 limit. However, a saturated switch can still forward a new flow (say between server s and server t) using the first wildcard rule in the routing table of the form $(s, *, p)$, $(*, t, p)$, or $(*, *, p)$. The degree of this switch is one in the residual graph used by MINNIE to compute Dijkstra. This decreases the computation time and the routing becomes very fast when the number of saturated switches is large (as the number of possible paths is then small). This is helpful as it may decrease the routing time of large topologies with a high number of flows.
Compression time. After having determined the path of a new flow and installed the rules along the path, we check if the size of one of the corresponding routing tables reaches the limit of 1000 rules. If so, MINNIE carries out a compression of the routing table. We plot in Figure 5b the average time to compress a routing table during the simulations for each group of topology. We see that even for the simulations of the largest topologies (pushed to their maximum with an all-to-all traffic of 6 millions of flows), the average compression time is below 16 ms. This corresponds to large routing tables dealing with 20 000 flows. A topology with around 1000 servers (1 million of flows in total) experiences an average compression time between 2 and 4 ms. As a typical example, we provide in Figure 6 the time needed to compress a switch for a BCube(32,1) and a 12-fat tree (432 servers) in function of the number of flows passing through it. For the 12-fat tree, the average compression time is 1.29 ms. For any switch, the first compression is done when reaching 1000 flows corresponding to 1000 forwarding rules (as aggregation rules are only introduced at the first compression). We then see that the second compression for a switch is done for around 2500 flows followed by compression when reaching 3000 to 4000 flows. These compression results show that previous compressions were efficient and that a large number of new flows are routed via aggregated rules. As for the two exceptions observed of tables compressed with around 18 000 flows[3], they correspond to one or two switches on which the paths are concentrated.

These time results allow to assume that the **impact of MINNIE on the controller load and on the flow delay will be limited** for these sizes of topologies. Note also that, when a new flow arrives, we choose to apply the compression module when the routing table size reaches the rule limit, but only after the new flow is routed. Thanks to this strategy, the delays experienced by the packets of the flow are not impacted by the compression carried out by the controller. These results are furthermore validated by

[3] Beware to distinguish the number of flows in the network from the number of rules. Here the number of rules per router is always below 1000 while the number of flows can be way higher.



(a) BCube(32, 1)

(b) 12 Fat Tree

Figure 6: Scatter plot of the time to compress a table as a function of the number of flows passing through the forwarding device.

running MINNIE on a data center testbed such as described in the following sections.

6 TestBed Description

We have built an experimental SDN-based data center testbed consisting of one HP 5400zl SDN capable switch (K15.15) with 4 modules installed – each module featuring 24 GigaEthernet ports – and 4 DELL servers. Every server possesses 6 quad-core processors, 32 GB of RAM and 12 GigaEthernet ports. On each server, we deployed 4 virtual machines (VMs), each VM with 8 virtual network interfaces. The 8 interfaces of each VM are further connected to one Open vSwitch (OVS) instance. The 4 OVS switches in each physical server are connected upstream to 4 (out of the 12) physical GigaEthernet ports in the physical server.

In one of the physical servers, we also deployed an additional VM hosting an SDN controller. To prevent the controller from becoming the bottleneck during our experiments, we configured it with 15 vCPUs (i.e., 15 cores) and 16 GB of RAM.

The topology of our data center network is a full 4-fat tree topology (see Figure 7), which consists of 20 SDN switches. To emulate those 20 SDN switches, we configured 20 VLANs on the physical switch. Each VLAN is an independent Openflow instance, making each VLAN an independent SDN-based switch. At every access switch, we connected two OVS switches to provide access to two VMs hosted in two different physical servers. Hence, each VM of a pod is hosted in a different physical server. In addition, each access switch hosts a single IP subnet with a total of 16 IP addresses corresponding to the 2 VMs, each with 8 interfaces, connected to it. Hence, in this network architecture, there are 8 subnets in total, with 16 different IP addresses (i.e. clients) per subnet. We detail in Section 6.1 the reason for choosing 16 clients per subnet.

The HP SDN switch can support a maximum of 65 536 (software + hardware) rules to be shared among the 20 emulated SDN switches. Software rules are handled in RAM and processed by the general-purpose CPU (slow path) while hardware rules are stored in the TCAM (fast path) of the switch. The number of hardware rules that can be stored per module in our switch being equal to 750, the total switch capacity is equal to 3000 hardware rules maximum. Those 65 536 (software + hardware) available entries are not equally distributed among the 20 switches as the concept of first flow arrived-first served policy is

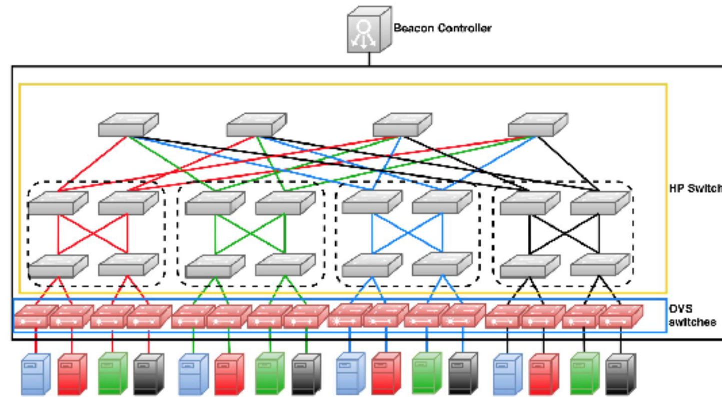


Figure 7: Our k=4 fat tree architecture with 16 OVS switches, 8 level 1, 8 level 2, and 4 level 3 switches.

used where the SDN rules are going to be installed on the HP switch in the order of arrival.

Regarding the controller, we use a Beacon [9] controller to manage all the switches (HP or OVS) in the data center. According to [26], Beacon features high performance in terms of throughput and ensures a high level of reliability and security.

In the remaining of this section, we justify our choice of 16 clients per access (level 1) switch and why we have decided to add virtual OVS switches between clients and level 1 switches.

6.1 Number of clients chosen for the experimentations

In our fat tree architecture, we can easily deduce the number of rules corresponding to a valid routing assuming that each VM talks to all other VMs in the data center that are not in its IP subnet. Considering no compression at all, one rule is needed for every flow passing through each switch along the path from a source to a destination. The set of flows that a switch "sees" depends on its level in the fat tree.

For any flow between two servers, the path goes through the level 1 switches to which each server is connected. Assuming n servers per level 1 switch ($n = 2$ in Figure 7), then each of the n servers connected to a level 1 switch communicates with the other $7 \times n$ servers in other subnets via outgoing and incoming flows. Overall, this represents $14n^2$ flows going through any level 1 switch.

The same argument can be used to find the number of flows for switches at other levels. This represents a total of $13n^2$ flows at each level 2 switch and $12n^2$ flows for a level 3 switch. In total, $264n^2$ rules are needed for the entire network.

In Figure 8, we compare the total number of rules with no compression at all, and with compression (obtained via simulation) on all switches. Without compression, only 15 clients per subnet can be deployed without running out of space in the forwarding table of our entire data center (65536 entries), while up to 36 clients can be deployed with the compression at the end. Therefore, Figure 8 explains our choice of installing 16 clients per subnet. Indeed, it is the first value for which the number of rules exceeds our total limit of number of rules (67584 rules) when no compression is achieved.

6.2 The need of level 0 OVS

OVS switches are used to make the controller aware of every new flow arriving in the fabric. Their routing tables are never compressed.

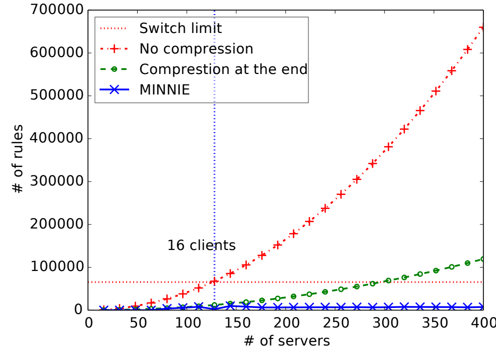


Figure 8: Total number of rules installed as a function of the number of servers, in a 4-fat tree configuration.

Without those switches, compressing at level 1 switches with MINNIE leads to non optimal route. This phenomenon can be explained by considering the case where clients would be directly connected to level 1 switches and MINNIE would be used at those switches. Suppose that a correct routing imposes at one of the access switches that to reach destinations d_1 and d_2, packets must be forwarded to port p_1 while for destination d_3, they should flow through port p_3. Without compression, we have three rules. Now suppose that MINNIE imposes that compression be done when the rules for destination d_1 and d_2 are present but the one for d_3 has not been installed yet. This leads to entries (s_1, d_1, p_1) and (s_1, d_2, p_1) being replaced by $(s_1, *, p_1)$. When packets from s_1 to d_3 are sent later, they will match the compressed forwarding rule and will reach d_3 using a longer path (or no path at all), as they will be forwarded to port p_1 and not p_3. In order to avoid this behavior, the controller should be contacted for every new flow in order to take the best routing decision for this flow. This is the role of the Openflow enabled OVS switches that we introduced. They enable the controller to perform compression with an exact knowledge of the set of active flows. The net result of using those OVS switches is to enable us to perform compression starting from level 1 switches, giving us more opportunity to use hardware rules at these switches. In the short version of this paper [25], we did not use level 0 OVS switches, and dealt with this problem by not compressing at level 1 switches, leading to lower compression ratios, and overloading of these switches.

Here, one could think that we have just migrated the problem from the edge devices to the physical server and that there is still a high number of non compressed rules. We believe however that this architecture represents an important step towards the solution of limited TCAM space because of the following reasons:

1. Virtualisation is a common service in modern data centers. Hence, a virtual switch (like Linux bridges or Open vSwitches virtual switches) are routinely used to provide network access to the virtual machines.

2. Data centers are closed networks under the control of a single team of network administrators that decide what is deployed or not at each server. Therefore, it is realistic to think that servers in data centers will all use Open vSwitches devices.

3. While at physical SDN-capable devices the TCAM size is a real problem (exceeding its capacity means that new rules will be "software rules" which will process packets by the slow path, leading to a drop of network efficiency), in virtual SDN-capable device, this is not a problem, since there is no slow path.

7 Experimental set-up

When the controller compresses a table, the MINNIE SDN application [4] will first execute the routing phase and then the compression phase. Hence, for the dynamic scenario, when a new flow must be routed with a new entry in the router, and if the threshold of X rule will be reached, the Xth entry is first pushed to such a switch (to allow the new flow to travel to the destination), and right after that, the compression is executed. Once the compression module is launched at the controller, a single OpenFlow command is used to remove the entire routing table from the switch. Then the new routes are sent immediately to limit the *downtime* period, that we define as the period between the removal of all old rules and the installation of all new compressed rules. When two or more switches need to be compressed at the same time, the compression is executed sequentially.

7.1 Experimental scenarios

We aim at assessing the performance of MINNIE with high number of rules and with high load. Those two objectives are contradictory in our testbed. Indeed, stressing the SDN switch in terms of rules, i.e., getting close to the limit of 65536 entries, imposes to have software rules. As software rules are handled, by definition, by the general purpose CPU of the switch, a safety mechanism has been implemented by HP to limit the processing speed to only 10 000 packets/s per VLAN. Assuming an MTU of 1500 bytes, we could not go beyond 120 Mb/s, shared between all ports in a VLAN. This is why we designed a second scenario where only hardware rules are used. In this scenario, we can fully use the 1 Gb/s link but we are limited to the 3000 hardware rules that have to be shared among the 20 switches. We thus built two scenarios to assess the performance and the feasibility of deploying MINNIE in real networks:

- **Scenario 1: Low load with (large number of) software rules**. This scenario enables to test the behavior of the switch when the flow table is full.
- **Scenario 2: High load with (small number of) hardware rules**. This scenario enables us to demonstrate that the network instability introduced by MINNIE remains negligible even when the switch transfer a load close to the line rate.

For each scenario, we consider three compression cases, which are similar to the simulation scenarios presented in Section 4.1.1:

- Case 1: No compression. We fill up the routing tables of the switches and we never compress them. This test provides the baseline against which we compare results obtained with MINNIE.
- Case 2: Dynamic compression at a fixed threshold. We set a threshold to the table size and compress whenever we reach this value. We extend the third scenario of the simulations by considering three thresholds values for scenario 1 (low load and software rules), namely 500, 1000 and 2000 entries, and also three values for scenario 2 (high load and hardware rules): 15, 20 and 30 entries.
- Case 3: Compression after installing the whole set of forwarding rules or when the forwarding table is full. This scenario illustrates the worst case and provides insights about the maximum stress introduced by MINNIE in the network. Indeed, in this case, we have the highest number of rules to be removed and installed after the compression executed by MINNIE which should be done as fast as possible.

While scenario 1 allows to test the scalability of MINNIE in terms of number of rules in real SDN equipments, this scenario might introduce, by default, an important jitter in the network because of the

[4] Available at: https://sites.google.com/site/nextgenerationsdndatacenters/our-project/minnie

usage of the general-purpose CPU to process the traffic. Moreover, an increase of the jitter during the compression events would suggest a non-negligible impact of the forwarding table replacement. Scenario 2 helps to better understand the impact of the compression and forwarding table replacement over the traffic. Since the traffic rate fills up to 75% of the access links, which is not enough to introduce congestion, and packets are processed by the ASIC, we expect to have a low jitter. Hence, any sudden increase of this last will immediately suggest an important impact of the compression mechanisms over the network stability.

7.2 Traffic pattern

We detail in this section how the two scenarios introduced in the previous section are actually implemented in our testbed.

7.2.1 Low load with software rules

In this scenario, the traffic is generated as follows: each client pings all other clients in every other subnet. This means that for each access switch, each of the 16 clients pings 112 other clients. There are no pings between hosts in the same subnet as we focus on the compression of classical IP-centric forwarding rules, which is used to route packets between different subnets, and not MAC-centric forwarding rules, as in legacy L2 switches.

We start with an initial client transmitting 5 ping packets to one other client. This train of 5 ICMP requests forms a single flow from the SDN viewpoint. We wait for this ping to terminate before sending 5 other different ping packets to another client, and so on, until all the 112 clients are pinged. When the first client finishes its pings series, a second client (hosted in the same VM) starts the same ping operation. Hence, the traffic is generated during all the experiment in a round-robin manner, among the 8 client of each VM. Moreover, VMs do not start injecting traffic at the same time. We impose an inter-arrival period of 10 minutes between them. Hence, VM 1 starts sending traffic at time zero, while VM 2 starts at minute 10, VM 3 at minute 20, and so on. This smooth arrival of traffic in the testbed is motivated by the fact that we do not wish to overload the physical switch with openflow events. Indeed, as stated in [19], commercial openflow switches can handle up to 200 events/s. Since in our testbed we have 20 switches, each one handling its own *flow_mod* (message for sending rules), *packet_out* (message with packet to be sent) and other events, the critical number of events can be easily reached.

The experiment of this scenario ran for almost 3.5 hours. All the rules are installed in the first 2 hours and 45 minutes.

7.2.2 High load with hardware rules

In this scenario, we used 1 client per VM so that the total number of rules installed (1056 total rules) is less than the hardware limit (3000 rules). Each VM starts a 50 Mbps ICMP traffic with the other clients in a round robin manner. After starting the first client machine, we wait for 75s and then start the outgoing connections for the second VM and so on, until all the machines establish connections with one other client. In this scenario, we have chosen 50 Mbps per connection in order to have a maximum of 800 Mbps load on a 1 Gbps link when all connections are established.

Each experiment of this scenario ran for 1 hour and all the rules are installed in the first 20 min. As mentioned earlier, all the rules were installed in hardware in order to reach high loads.

Level	No Comp	Comp 500	Comp 1000	Comp 2000	Comp full
level 1	3452	752	761	790	802
level 2	3233	618	649	672	717
level 3	3014	97	97	97	97
total	65 535	11 346	11 667	12 087	12 542

Table 4: Average number of SDN rules installed in a virtual switch at each level

Level	Comp 500	Comp 1000	Comp 2000	Comp full
level 1 (8 switches)	79%	78.75%	77.95%	77.61%
level 2 (8 switches)	81.43%	80.51%	82.14%	78.45%
level 3 (4 switches)	96.84%	96.84%	96.84%	96.83%
total (20 switches)	83.21%	82.19 %	81.55 %	81.44%

Table 5: Average percentage of SDN rules savings at each level

8 Experimental Results

In this section, we validate MINNIE through experiments on the fat tree testbed described in Section 6. We present the results obtained for each experimental scenarios in terms of number of rules of the routing tables, compression time, delay, packet loss and controller load.

8.1 Scenario 1: Compression with low load and software rules

8.1.1 Number of rules with/without compression

As explained in Section 6.1, in this scenario and without compression, the limit of 65 536 entries in our HP switch is reached. On the other hand, compressing the table with MINNIE allows to install all the required rules without reaching the limit when compressing at a given threshold (500, 1000 or 2000 entries) or when the flow table is full. Indeed, as shown, in Table 4, the total number of installed rules does not exceed 13 000 rules in all compression cases. This represents a total saving higher than 80% of the total forwarding table capacity (Table 5) with a saving larger than 96% at the third level and a minimal saving over 76%.

Figure 9 depicts how the number of rules evolves over time with and without compression. Please, note that such a figure takes into account the total number of forwarding rules in the network, including both Open vSwithes and the HP switch. The number of rules increases at the same pace in all 3 scenarios during the first 30 minutes. When the compression is triggered, the number of rules decreases. Later, for compression at 500 and 1000 entries, the number of rules increases at a lower pace than in the non compression case. This is because (i) the controller has installed some wildcard rules and so no new rules at level 1, 2 or 3 need to be installed for new flows, and (ii) other compression events are triggered. We further notice here that the presence of wildcard rules also explains the difference between the compression when the forwarding table is full and the compression with fixed thresholds. This is inline with the results of Section 4.2 where we observed that the presence of wildcard rules in the routing tables influences the routing as the new incoming flows will follow these paths in priority. Even though the difference between dynamic compression and compression at the end is more pronounced for networks with larger number of servers (see Figures 4), the phenomenon can already be observed in the testbed. In Fig 8, we can see how this difference between the two curves is evolving with the number of servers.

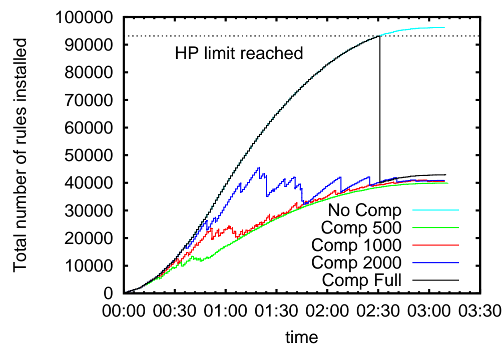


Figure 9: Total number of rules installed in the whole network

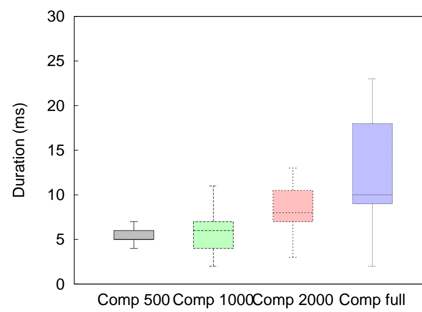


Figure 10: Average duration of compression period.

Threshold	No Comp	Comp 500	Comp 1000	Comp 2000	Comp full
# of compressions	NA	16 594	95	28	20
% pkt loss	6.25×10^{-6}	0.003	5.65×10^{-4}	2.83×10^{-5}	3.7×10^{-4}

Table 6: Total number of compressions and packet loss rate.

8.1.2 Compression time

Figure 10 shows the compression time seen by the controller, which consists of the computation time of compressed rules (already analyzed in Section 5.5), the removal of the current forwarding table, the formatting of the compressed rules to the OpenFlow standards, and the injection of the new rules to the switch.

We notice that the compression time per switch remains in the order of a few milliseconds. Indeed, compression takes about 5 ms (resp. 7 ms) for compression at 500 and 1000 entries (resp. 2000 entries). Even the worst case – compressing when the table is full – represents less than 18 ms for most of the switches with a median at 9ms. Moreover, in this latter case, sequentially compressing all switches requires no more than 152 ms. This compression period is mainly due to the time needed to delete all the routing table and install all the new rules in the switch. Indeed, the time needed to compute the compressed routing table is negligible as observed in Section 5.5 Figure 5.It is important to note here that the code used to compute the compressed tables is the same in our simulations and experiments.

8.1.3 SDN control path

In the SDN paradigm, the controller to switch link is a sensitive component as the switch is CPU bounded and cannot handle events at a too high rate. Figure 11 represents the network traffic between the switch and the controller in the different scenarios. We can observe that the load increases highly when the switch limit in terms of number of software+hardware rules is reached and we do not compress the routing tables. After time t=2:30, the limit is reached and for every packet of every new flow, each switch along the path has to ask the controller for the output port. These traffic peaks vanish when we compress the routing tables for the 1000 and 2000 limits or for the case of compression when full. As for the compression at 500 scenario we notice the occurrence of high peaks after the first hour. They result from successive compression events (over 16 000 in our experiments as can be seen in Table 4) that are triggered by any new packet arrival. Indeed, in this scenario, most of the switches will perform a compression for every new flow, since the total number of rules after compression remains higher than the threshold.

To understand the impact of the control plane on the data plane, we have to look at three key metrics that we detail in the following sections: (i) the loss rate for all scenarios; (ii) the delay of the first packet of new flows that should be higher when there is no compression (at least after t=2:30) or at compression at 500 and (iii) the delay of subsequent packets (packets 2 to 5) that should be larger for the case of no compression when the table is full. We ruled out a precise study of the loss rate as the load in this section is low. We report in Table 4 the loss rates observed for all scenarios. Though there exist some significant differences between the different scenarios, the absolute values are fairly small. We therefore focus on delays hereafter.

8.1.4 New rules installations: Impact on first packet delay

The first packet delay provides insights on the time needed to contact the controller and install the rules when a new flow arrives. Indeed, the round trip delay seen by the first packet of a new flow includes the network propagation delay, the queuing delay, and the time needed by a switch to obtain a new rule.

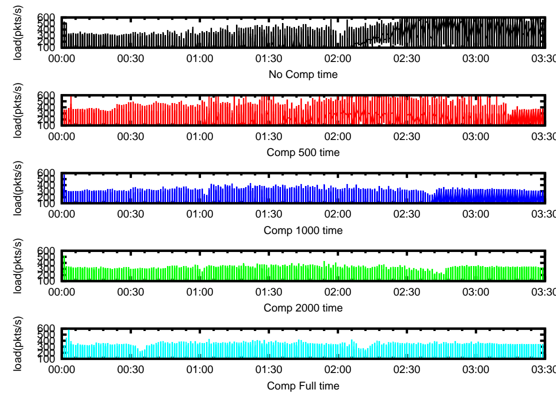
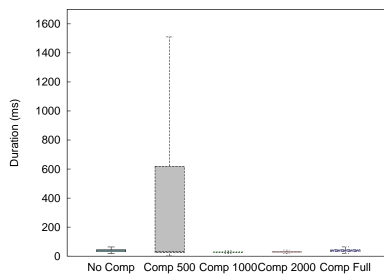
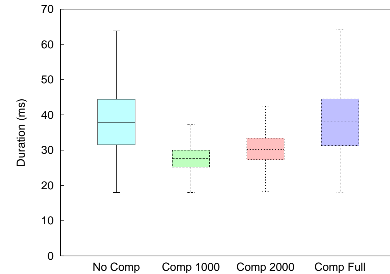


Figure 11: Network traffic between the switches and the controller.



(a) First packet delay boxplot with compression 500



(b) First packet delay boxplot without compression 500

Figure 12: First packet delay boxplot

We observe in Figure 12 that for the scenarios compression at 1000 rules and compression at 2000 rules, the first packet delay ranges from 25ms to 35ms. This increase as compared to subsequent packets of the same flow- which can reach a factor of 10 as we will in the next section- highlights the high price to pay to obtain and install a forwarding rule in software. The results can significantly worsen if the controller is frequently modifying the forwarding rule, like in the compression at 500 rules case. Indeed, for that special case, the third quantile reaches up to 600ms for the first packet delay.

Surprisingly, the cases without compression and compression at the table limit lead to similar results. Compressing when the table is full should intuitively lead to better performance as in a number of cases, a limited number of new rules need and can be installed as compared to the no compression case. However, in our tests, the table becomes full after 2 hours and 30 min of experiment (out of 3 hours). Hence the similarity of results in Figure 12. In fact, when the table is full, the impact is striking, as can be seen in the time series of Figure 13a, which shows the evolution of the first packet delay per new flow when no compression is executed. Indeed, after 2:30 hours - when the table is full- we can observe a jump in the delay for no compression while when compressing at the table limit the trend is the opposite and the delay

decreases (Figure 13e) after compression. As for the case of compression at 500, the first packet delay features a chaotic behavior (Figure 13b) due to its high compression frequency as expected. Regarding the scenarios of compression at 1000 (Figure 13c) and compression at 2000 (Figure 13d), the benefits of compressing periodically are stricking: the first packet delay shows a constant trend during the whole experiment.

Eventually, note that the results obtained here are impacted by the fact that we use software rules, which increases the delay to install rules. Results of the experiments using hardware (i.e. TCAM) rules exclusively are provided in Section 8.2.

8.1.5 Subsequent packets delay

As explained previously, we expect to observe higher delays for subsequent packets for the case of no compression (when the table is full) and also possibly for the case of compression at 500 as the switches have to reinstall new rules at a high frequency.

In our experiments, the delay seen by packets 2 to 5 of each flow is shorter than 4ms most of the time for scenarios without compression, compression at 1000, compression at 2000 and compression at the forwarding table size limit, as we can see in Figure 14. Compression at 500 is slightly different (the third quartile reaches up to 5ms), highlighting the negative impact of the high frequency of compression events on the data path of the switches.

Figure 14 aggregates all the results together and we have again to resort on the time series to observe specific effects. When all needed forwarding rules are successfully installed and the compression frequency is low (which is the case for compression at the limit, compression at 1000 and compression at 2000), the delay of packets 2 to 5 is consistently comprised between 2ms and 6ms (Figures 15d, 15e and 15f).

Without compression, while most of the packets experience a delay between 2ms and 6ms before the table limit, all new incoming packets will see a delay equal or higher than 40ms afterwards (Figure 15b). As for the case of compression with small table limit (500 rules), we remark in Figure 15c a time interval between 1:45 hour and 2:15 hour, where the delay increases suddenly from 2 ms to 100 ms. This is because some switches are unable to reach a forwarding table smaller than 500 rules even after compression, and hence, the controller executes a compression after every new flow arrival. After time 2:15, the frequency of new incoming flows that need to be installed decreases (Figure 13b), leading to a stabilization of the delay.

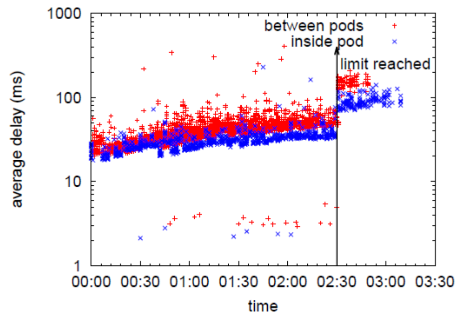
8.2 Scenario 2: Compression with high load and hardware rules

We have so far investigated the behavior of MINNIE in an environment where the flow table can be full. The latter scenario involves the use of software rules and thus the slow path of our HP switch.

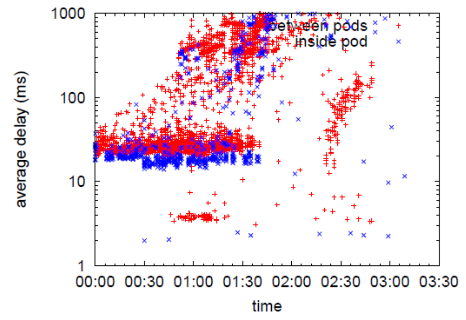
We now turn our attention to the case where the load on the data plane is as high as 80%. This entails using hardware rules only and we are limited to 3000 such rules with our HP switch, shared among the 20 switches of our k=4 fat tree topology. The experiments in this section are consequently performed with 1 client per access switch (16 clients in total) and an all-to-all traffic pattern with 50Mbps per flow. Since the flows are equally spread across the network links, we have a link utilization of about 75%.

As expected, the first packet round trip delay decreases to around 1ms, while packets 2 to 5 experience a round trip delay of around 0.55ms [5]. Even though it is not graphically shown, the compression duration, in all scenarios is equal to 1ms only. **We further noticed no packet losses and no drastic effects on delay even during compression events, which proves that MINNIE is a viable and realistic solution**. Indeed, the maximum variation of delay between the delays of no compression and all compression

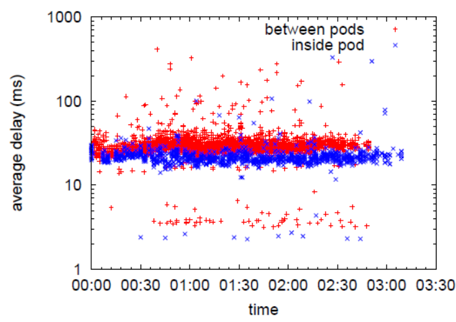
[5] A direct comparison between these delays and the one for the low load and software rules scenario is not straightforward. Section 9 will present a fair comparison of these two modes.



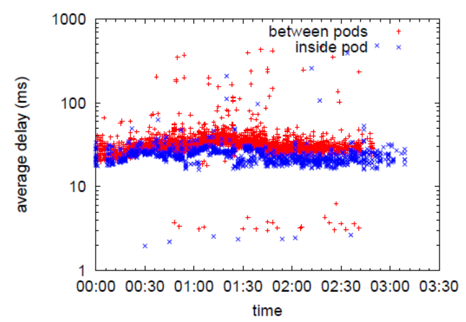
(a) Without compression



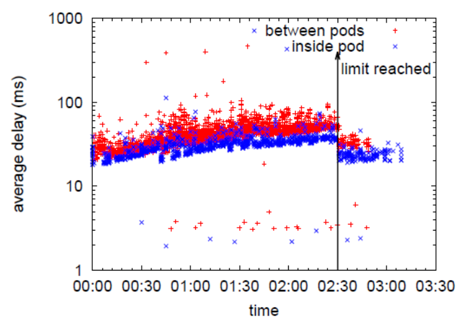
(b) Compression 500



(c) Compression 1000



(d) Compression 2000



(e) Compression when full

Figure 13: First packet average delay with low load

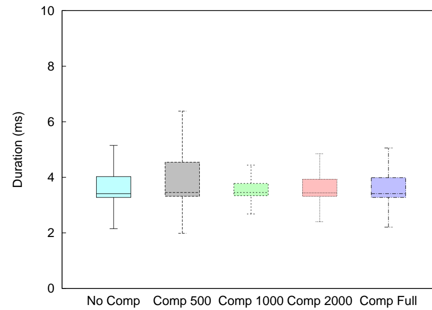


Figure 14: Average packet's delay boxplot for pkts 2 to 5

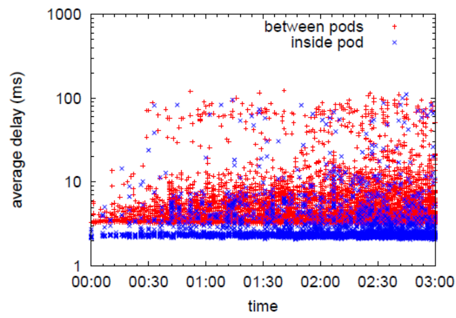
scenarios is less than 0.1 ms, a value which might be observed even in non-SDN networks (see Figure 16).

The compression ratio in Table 7 demonstrates that even with a low number of rules, MINNIE can achieve a high compression ratio, over 70%. Figure 17 which represents the evolution of the forwarding table size for all cases – no compression, compression at 15, 20, 30 and when full (after installing all the needed rules)– highlights that MINNIE maintains a similar low number of rules in all compression scenarios.

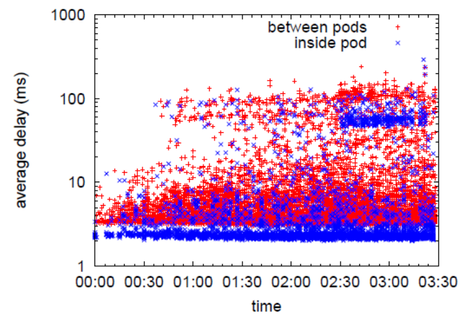
Level	Compression 15	Compression 20	Compression 30	Compression when full
level 1 (8 switches)	76.56%	75.66%	75%	72.76 %
level 2 (8 switches)	75.48%	73.31%	71.87%	69.71 %
level 3 (4 switches)	76.04%	76.56%	74.47%	73.95 %
total (20 switches)	76.04%	74.9 %	73.67 %	71.78 %

Table 7: Average percentage of SDN rules savings at each level

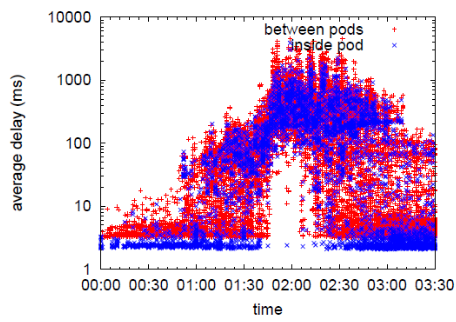
A last question that we aim at investigating is the impact of compression on TCP connections. The high load scenario is especially relevant as data centers are in general operated at high loads. The variation of the round trip delay of most of the packets is less than 0.1ms (Figure 16) for compression at 20 entries with the highest variability. For compression at 20 entries and during the first 20 minutes of the experiment (compression events occur during that period), the minimum and maximum round trip delays between servers in the same pod is around 0.4ms and 0.6ms respectively, while the minimum and maximum round trip delays between servers in different pods is around 0.55ms and 0.8ms respectively (see Figure 18). Those observed delays will not produce any problem to TCP connections. Indeed, the minimum RTO value (the time needed to trigger a TCP timeout and retransmit a non Acked packet), is equal to 200ms in Linux systems (and defined to be 1 second in the RFC 2988 [24]), which is far from our observed delays (lower than a millisecond). A recent draft submitted to the TCPM Working Group [5] appeals for a decrease of the minimum RTO value to 10ms. Once again, the maximum delay observed during the compression events is still far from that proposed minimum RTO. Hence compression operations should not lead to any spurious TCP time out. Note eventually that results obtained in the simulations on the computational time (Figure 5 of Section 5) confirm that the impact of MINNIE on the delay experienced by the packets of the flow will be limited for all the studied topologies.



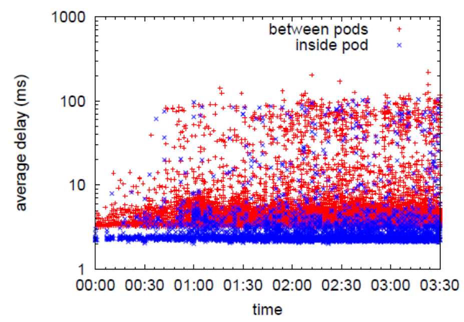
(a) Without compression 7 IPs



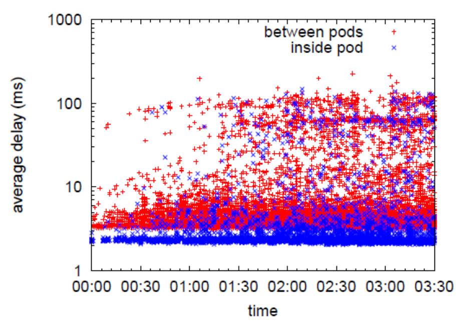
(b) Without compression 8 IPs



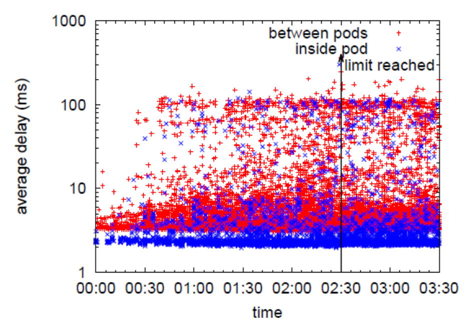
(c) Compression 500



(d) Compression 1000

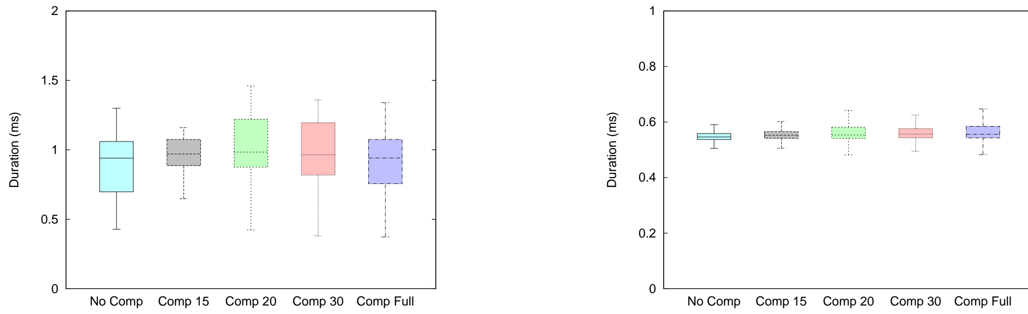


(e) Compression 2000



(f) Compression when full

Figure 15: Average packet delay of pkts 2 to 5 with low load



(a) First packet's delay

(b) Average packet's delay except first packet

Figure 16: Packet delay boxplot under high load.

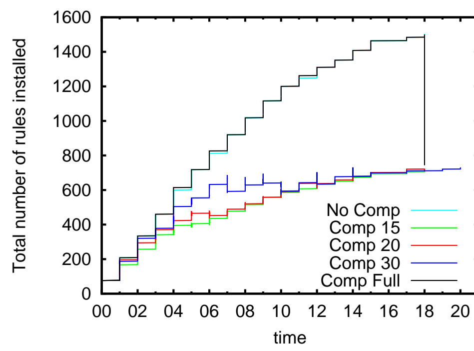


Figure 17: Total number of rules installed in the network

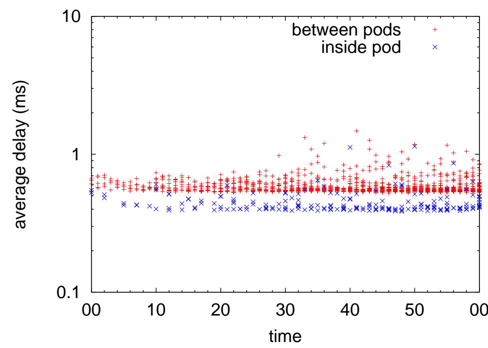


Figure 18: High load and hardware rules : Delay of packets 2 to 5 - Compression at 20 entries

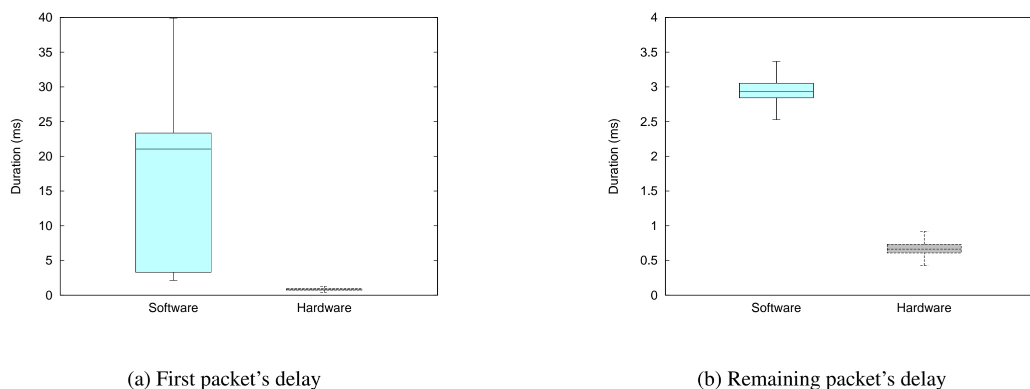
9 Software vs. hardware rules

So far, we have seen that relying only on the ASICs of the switch to forward the traffic provides better results, in terms of delay and jitter, than using the general purpose CPU for such a task. Hence, one question naturally rises at this point: what is the real impact of the *slow path* on the switch performance?

Assessing the difference between using hardware and software rules by comparing the results of Sections 8.1.1 and 8.2 is difficult as the number of rules is different from one scenario to the other. For this reason, we devised a third scenario where we compare the performance of software and hardware rules using both, the same number rules and the same traffic load, in all cases.

In this experiment, we have one client per access switch, and each flow is composed by a train of 5 ICMP request / reply packets, which is the default behavior of the `ping` command. With this configuration, we can observe in Figure 19a that installing rules in software increases the first packet delay by a factor of 20 from a median of 1ms to 20ms as compared to hardware rules. The average matching delay of the remaining packets (Figure 19b) features a 6-fold increase in software as compared to hardware (3ms compared to 0.5ms).

The results obtained with these experiments thus confirm the large discrepancy in terms of average delay results between Sections 8.1.1 and 8.2. **They further justify the necessity of using only TCAM memory, which can be better exploited thanks to the compression executed by MINNIE.**



(a) First packet's delay

(b) Remaining packet's delay

Figure 19: Packet delay boxplot

10 Discussion

The results obtained in Sections 5 and 8 via simulations and experiments respectively demonstrate the feasibility and efficiency of MINNIE. A next natural question is whether they can be used to pick operational parameters for MINNIE in the general case, i.e., (i) for any application workload (ii) for any data center topology.

Let us first consider the workload issue. We have used an all-to-all traffic pattern which arguably constitutes a worst case in terms of traffic workload that an application could possibly generate in the network. This workload also constitutes a worst-case for the SDN control plane as it entails establishing the maximum number of connections between switches. Indeed, a client connected to an access switch establishes a connection with every other machines in the other access switches, i.e., a machine talks to 112 other IPs. In an operational network deployment, it is reasonable to admit that (i) an access switch

hosts a small number of IP subnets (VLAN) and (ii) SDN rules are installed on an IP subnet basis and not on a flow basis. Thus, the results in Section 8 can be interpreted as if we had 16 IP subnets per access switch and one rule for every possible inter-VLAN connection. From an SDN perspective, having more machines per VLAN would thus not increase the number of rules. The first experimental scenario *low load and software rules* thus demonstrates that MINNIE is able to limit the number of SDN rules per switch in such a way that only hardware rules will be used. In other words, the results obtained in Section 8 would hold for every application workload and a 4-fat tree topology even if we increase the number of machines per VLAN.

To assess the dependence of the results on the exact topology, one can rely on the results obtained in Section 5 that demonstrate that for more servers (which, again, can be interpreted as more VLANs) and a variety of topologies, compressing SDN routing tables reduces drastically the number of rules needed to perform routing (see Figures 2 and 4). Combining the load-balanced routing with the dynamic compression of MINNIE leads to small forwarding tables in such a way that they fit into typical TCAMs (see Table 2). **In summary, adopting compression with a limit at** 1000 **or** 2000 **should be valid for a wide spectrum of workload and data center topologies**.

A last dimension that we have not explored during our tests is the burstiness of arrival of flows that could lead to stress the switch-controller communication, and hit the limit of a few hundreds events/s that the switch is able to sustain. This could be the case of an application that generates a lot of requests towards a large set of servers at high rate. In this situation, MINNIE could help alleviating the load on the controller. Indeed, the sooner one compresses the flow table, the more likely we are to install rules that will prevent the switch from querying the controller for a rule for every new connection. One could argue that compressing entails complete modification of the flow table at the switch, ie. a large number of events (deletion, insertion) related to the management of the table. However, in OpenFlow, those events can be grouped together: all insertions can be sent at once to the switch. **In summary, MINNIE should also help alleviating the stress of the switch-controller channel in case of flash-crowds of new connections.**

11 Conclusion and Future Work

SDN enables to formulate complex forwarding rules. However, such a flexibility requires expensive and limited in size TCAMs. Even if the capacities of TCAMs is expected to increase in the near future, we still have to pay a specific attention to the controller-switch path that should not lead to overload hardware SDN switches that are CPU bounded. There is thus a need to reduce as far as possible the number of rules that each switch has to manage.

In this paper, we have introduced MINNIE, which aims at computing load-balanced routes, and at compressing SDN routing tables using aggregation by the source, the destination and with the default rule. We have investigated through numerical experiments the versatility of MINNIE on a variety of data center topologies and demonstrate that it can handle close to a million of flows with no more than 1000 rules per switch.

Numerical results have been complemented with experiments on a testbed emulating a 4-fat-tree topology. Those experiments have confirmed the ability of MINNIE to drastically reduce the number of rules to manage with no noticeable negative effect on delay or losses.

References

[1] NEC univerge PF5240 and PF5820. http://www.openflow. org/wp/switch-nec/.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[3] D. L. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang. Compressing Rectilinear Pictures and Minimizing Access Control Lists. In *ACM-SIAM SODA*, pages 1066–1075, 2007.

[4] S. Banerjee and K. Kannan. Tag-in-tag: Efficient flow table management in sdn switches. In *CNSM*, pages 109–117, 2014.

[5] S. Bensley, L. Eggert, D. Thalerand P. Balasubramanian, and G. Judd. Microsoft's Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters draft-bensley-tcpm-dctcp-05. Internet-Draft, July 2015.

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, pages 99–110. ACM, 2013.

[7] W. Braun and M. Menth. Wildcard compression of inter-domain routing tables for openflow-based software-defined networking. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 25–30, Sept 2014.

[8] R. Cohen, L. Lewin-Eytan, J.S. Naor, and D. Raz. On the effect of forwarding table size on sdn network utilization. In *INFOCOM*, pages 1734–1742. IEEE, 2014.

[9] David Erickson. The beacon openflow controller. In *HotSDN*, pages 13–18. ACM, 2013.

[10] Frédéric Giroire, Frédéric Havet, and Joanna Moulierac. Compressing two-dimensional routing tables with order. In *INOC*, pages 1–8, 2015.

[11] Frédéric Giroire, Joanna Moulierac, and T Khoa Phan. Optimizing rule placement in software-defined networks for energy-aware routing. In *GLOBECOM*, pages 1–6. IEEE, 2014.

[12] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39:4, pages 51–62. ACM, 2009.

[13] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: A high performance, server-centric network architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 39(4):63–74, August 2009.

[14] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A scalable and fault-tolerant network structure for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):75–86, August 2008.

[15] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 15–28, Berkeley, CA, USA, 2015. USENIX Association.

[16] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *CoNEXT*, pages 13–24. ACM, 2013.

[17] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *INFOCOM*, pages 545–549. IEEE, 2013.

[18] K. Kannan and S. Banerjee. Compact TCAM: Flow Entry Compaction in TCAM for Power Aware SDN. In *ICDCN*, pages 439–444, 2013.

[19] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[20] C. R. Meiners, A. X. Liu, and E. Torng. TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs. In *IEEE/ACM Transaction in Networking*, pages 490–500, 2010.

[21] C. R. Meiners, A. X. Liu, and E. Torng. Bit Weaving: A Non-prefix Approach to Compressing Packet Classifiers in TCAMs. In *IEEE/ACM Transaction in Networking*, pages 488 – 500, 2012.

[22] J. Moy. OSPF Version 2. RFC 2328 (Standard), 1998.

[23] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. OFFICER: A general Optimization Framework for OpenFlow Rule Allocation and Endpoint Policy Enforcement. In *INFOCOM*, pages 478–486. IEEE, April 2015.

[24] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), November 2000. Obsoleted by RFC 6298.

[25] Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, Dino Martín Lopez Pacheco, Joanna Moulierac, and Guillaume Urvoy-Keller. Too many SDN rules? Compress them with MINNIE. In *GLOBECOM* , pages 1–6. IEEE, December 2015.

[26] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. Advanced study of SDN/openflow controllers. In *CEE-SECR*, pages 1:1–1:6. ACM, 2013.

[27] Michael Theobald, Steven M. Nowick, and Tao Wu. Espresso-hf: A heuristic hazard-free minimizer for two-level logic. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 71–76, New York, NY, USA, 1996. ACM.

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399