

Poster: Scaling Data Plane Verification with Throughput-Optimized Atomic Predicates

Dong Guo¹ Jian Luo, Kai Gao² Y. Richard Yang³

¹Tongji University ²Sichuan University ³Yale University

Abstract

Atomic predicate is a key enabler for data plane verification, usually with BDD as underlying data structure. However, BDD libraries hinder verification from real-time because they don't scale well, especially in large-scale networks.

Crux of scaling verification

Small changes may have significant impacts. For large-scale networks, data plane verification (DPV) system usually should accommodate millions of FIB rules, and handle FIB updates that may overlap with existing rules at the same scale. Thus, there is a strong demand for a scalable BDD infrastructure to achieve scalable DPV.

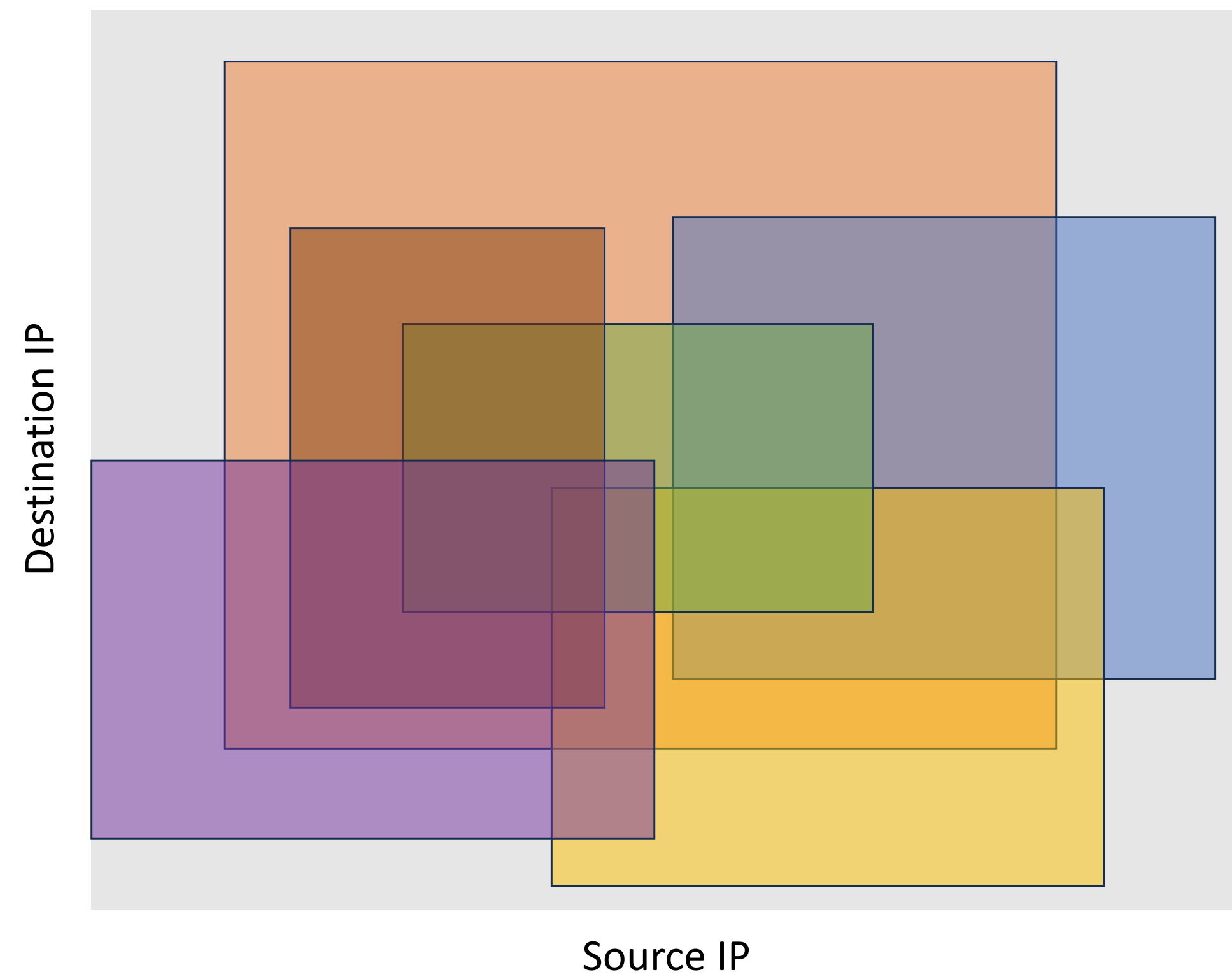


Figure 1. How small changes lead to significant impacts.

Bottleneck lies in BDD package.

The scalability of DPV systems is restricted by the scalability of BDD operations for two reasons:

1. Previous work Flash have shown that BDD operations can account for **the majority of the verification time** (i.e., the number of BDD operations reaches the order of $O(10^9)$ and takes over 99% of the overall time).
2. Many parallelizable processes are hindered due to the **absence of parallelism support**. In cases of thread-safe packages, they either adopt expensive synchronizations or unsuitable scheme. In root, existing BDD packages are **not designed for throughput**.

Table 1. Publicly available BDD libraries.

Name	Lang	Thread-safe	Scalability	Synchronization	Execution
BuDDy	C/C++	N	Poor	User-lock	RtC
CUDD	C++	N	Poor	User-lock	RtC
JDD	Java	N	Poor	User-lock	RtC
BeeDeeDee	C++	Y	Medium	Mutex-lock	RtC
PJBDD	Java	Y	Medium	Lock-free	FJ
Sylvan	C++	Y	Medium	Lock-free	FJ
HermesBDD	C++	Y	High	Lock-free	FJ
TOBDD	C++	Y	High	Lock-free	RtC

Designs of Throughput-Optimized BDD

TOBDD is designed to achieve high throughput parallel BDD operations by leveraging the multi-core capabilities of modern hardware.

- **Reschedulable Thread-pool** to eliminate the overhead of frequent thread creation and destruction,
- **Lock-free Node Table and Cache**. Inside it leverages CAS to achieve lock-free and dynamic linked lists to avoid resizing
- **Run-to-Completion** to avoid the overhead of interruption.
- **Commutative Hash Function** to reduce cache misses.

Use case: Scaling DPV with TOBDD

We implement TOBDD^a and a throughput-optimized version of Flash called TOFlash in C++. We compare the data plane model construction time of TOFlash with our previously implemented Flash artifact on various datasets. Figure 2 shows the evaluation results. We can see that

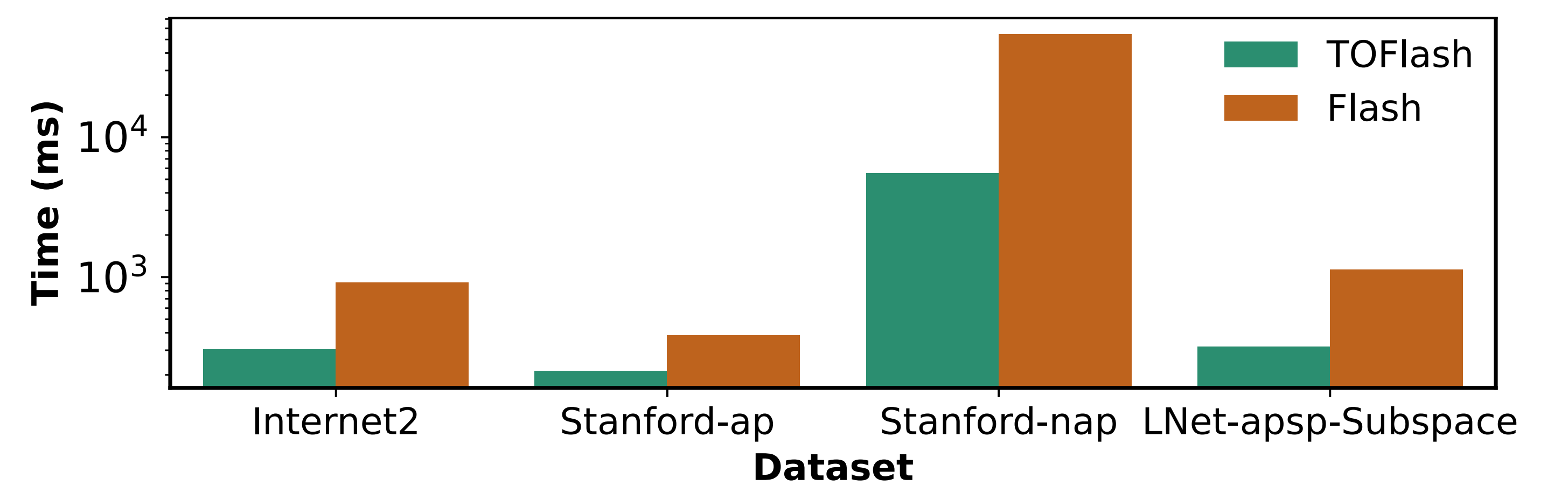
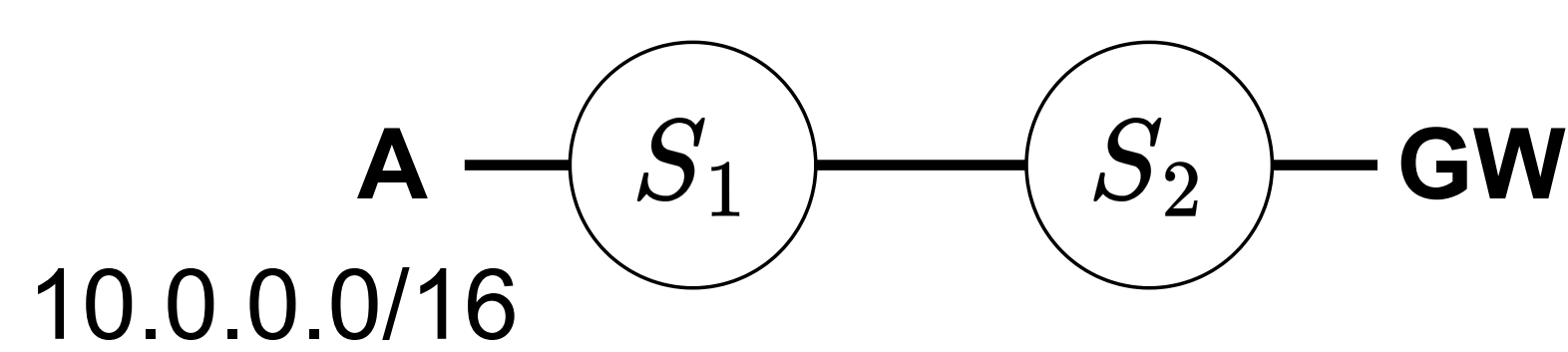


Figure 2. TOFlash achieves 2-10x improvement compared with the baseline.

^a<https://github.com/guodong/tobdd>

Initial Data Plane

\mathcal{R}		
	R_1	R_2
10.0.0.0/16	A	S_1
0.0.0.0/0	S_2	GW



Event:

Add a new policy that traffic in IP prefix 10.0.0.0/24 and 10.0.0.0/8 to subnet A uses path $S_2 - S_1$

$\Delta \mathcal{R}$		
switch	match	action
S_1	+10.0.0.0/24	A
S_1	+10.0.0.0/8	A
S_2	+10.0.0.0/24	S_1
S_2	+10.0.0.0/8	S_1

$\Delta \mathcal{R}'$		
switch	predicate	action
S_1	$+p_3$	A
S_1	$+p_2$	A
S_2	$+p_3$	S_1
S_2	$+p_2$	S_1

$\Delta \mathcal{M}'$		
switch	predicate	action
S_1	p_3	A
S_1	$p_2 \wedge \neg p_1$	A
S_2	p_3	S_1
S_2	$p_2 \wedge \neg p_1$	S_1

$\Delta \mathcal{M}''$		
switch	predicate	action
S_1	$p_3 \vee (p_2 \wedge \neg p_1)$	A
S_2	$p_3 \vee (p_2 \wedge \neg p_1)$	S_1

\mathcal{M}		
	S_1	S_2
p_1	A	S_1
$p_0 \wedge \neg p_1$	S_2	GW

$\Delta \mathcal{M}$		
predicate	S_1	S_2
$p_2 \vee (p_3 \wedge \neg p_1)$	A	S_1

\mathcal{R}'		
	R_1	R_2
10.0.0.0/24	A	S_1
10.0.0.0/16	A	S_1
10.0.0.0/8	A	S_1
0.0.0.0/0	S_2	GW

p_0 : dstip = 0.0.0.0/0
p_1 : dstip = 10.0.0.0/16
p_2 : dstip = 10.0.0.0/8
p_3 : dstip = 10.0.0.0/24

\mathcal{M}'		
	S_1	S_2
p_2	A	S_1
$p_0 \wedge \neg p_2$	S_2	GW

\mathcal{M}'		
	S_1	S_2
p_3	A	S_1
$p_1 \wedge \neg p_3$	A	S_1
$p_2 \wedge \neg p_1$	A	S_1
$p_0 \wedge \neg p_2$	S_2	GW

→ Parallel execution (map) → Parallel aggregation (reduce)

Final Data Plane Model

Figure 3. An example of utilizing parallelism in DPV