Dynamic Scheduling of Network Updates

Xin Jin

Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan,
Ming Zhang, Jennifer Rexford, Roger Wattenhofer

Microsoft Research

PRINCETON UNIVERSITY

PURDUE

ETH Zürich

Dynamic Scheduling of Network Updates

Xin Jin

Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, **Ratul Mahajan**,

Ming Zhang, Jennifer Rexford, **Roger Wattenhofer**

Ratul Mahajan

C.-Y. Hong, S. Kandula, **R. Mahajan**, M. Zhang, V. Gill, M. Nanduri, and **R. Wattenhofer**, "Achieving high utilization with software-driven WAN," in ACM SIGCOMM, 2013.

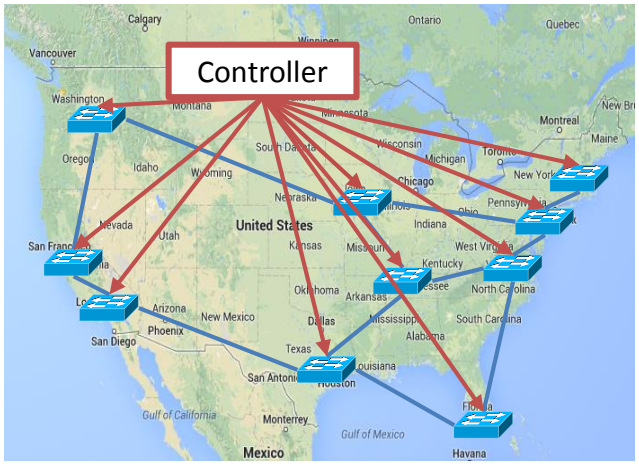
Roger Wattenhofer

R. Mahajan and **R. Wattenhofer**, "On consistent updates in software defined networks," in ACM SIGCOMM HotNets Workshop, 2013.

Microsoft Research

SDN: Paradigm Shift in Networking

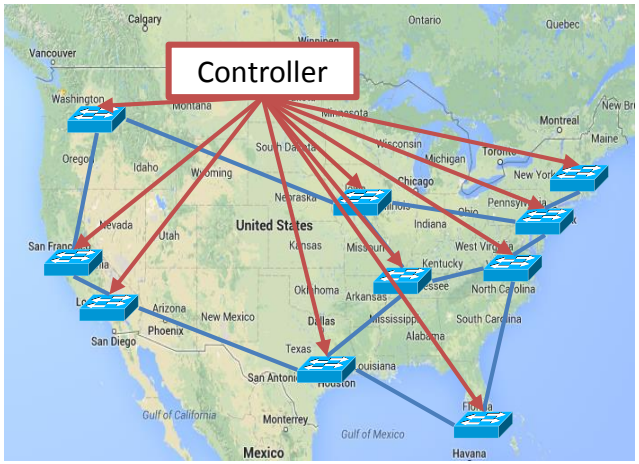
- Direct, centralized updates of forwarding rules in switches

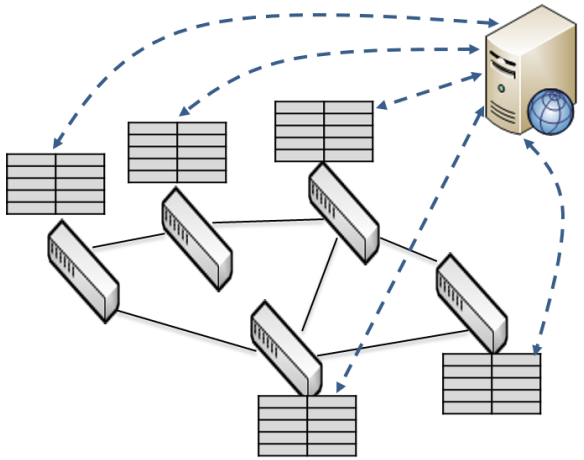


- Many benefits
  - Traffic engineering [B4, SWAN]
  - Flow scheduling [Hedera, DevoFlow]
  - Access control [Ethane, vCRIB]
  - Device power management [ElasticTree]

Update the data plane state

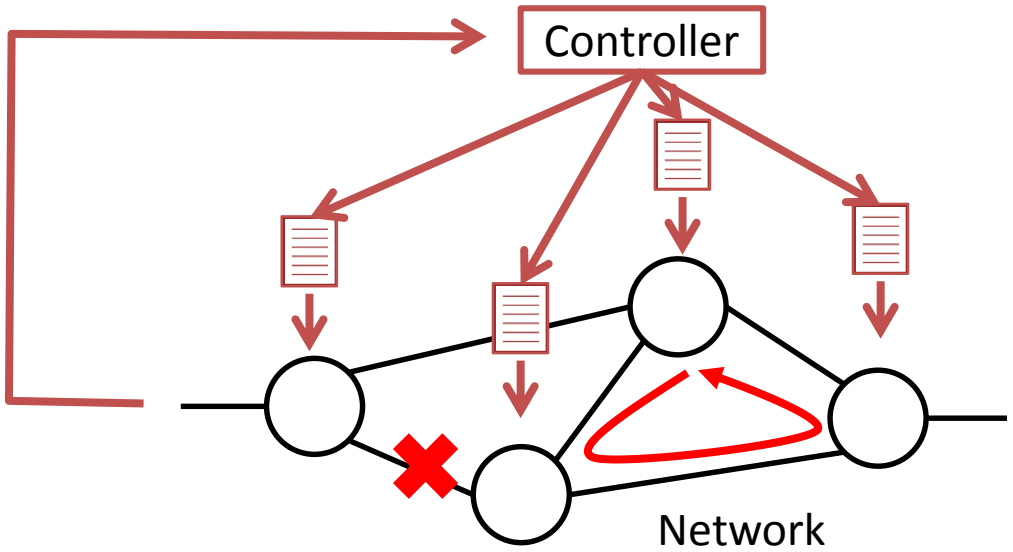
- Scenarios
  - Periodically traffic engineering
  - Failure recovery
- The state
  - a set of rules
  - how switches forward packets



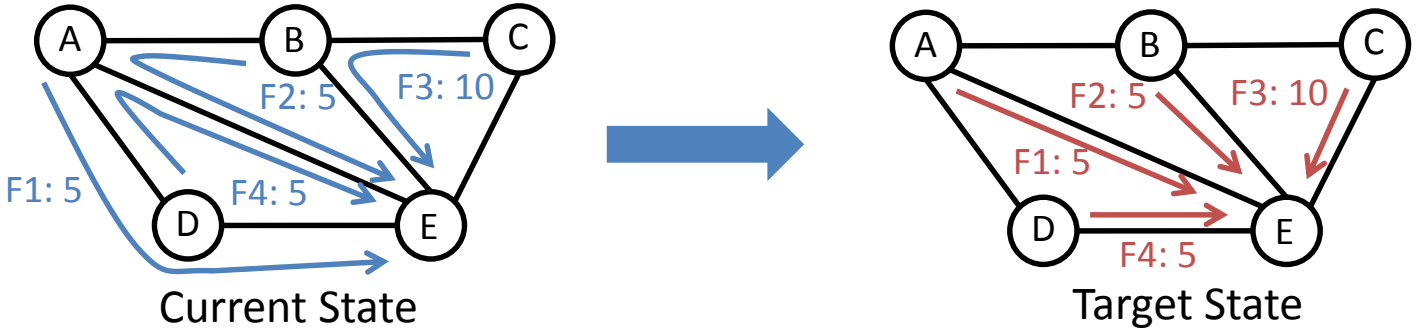


Network Update is Challenging

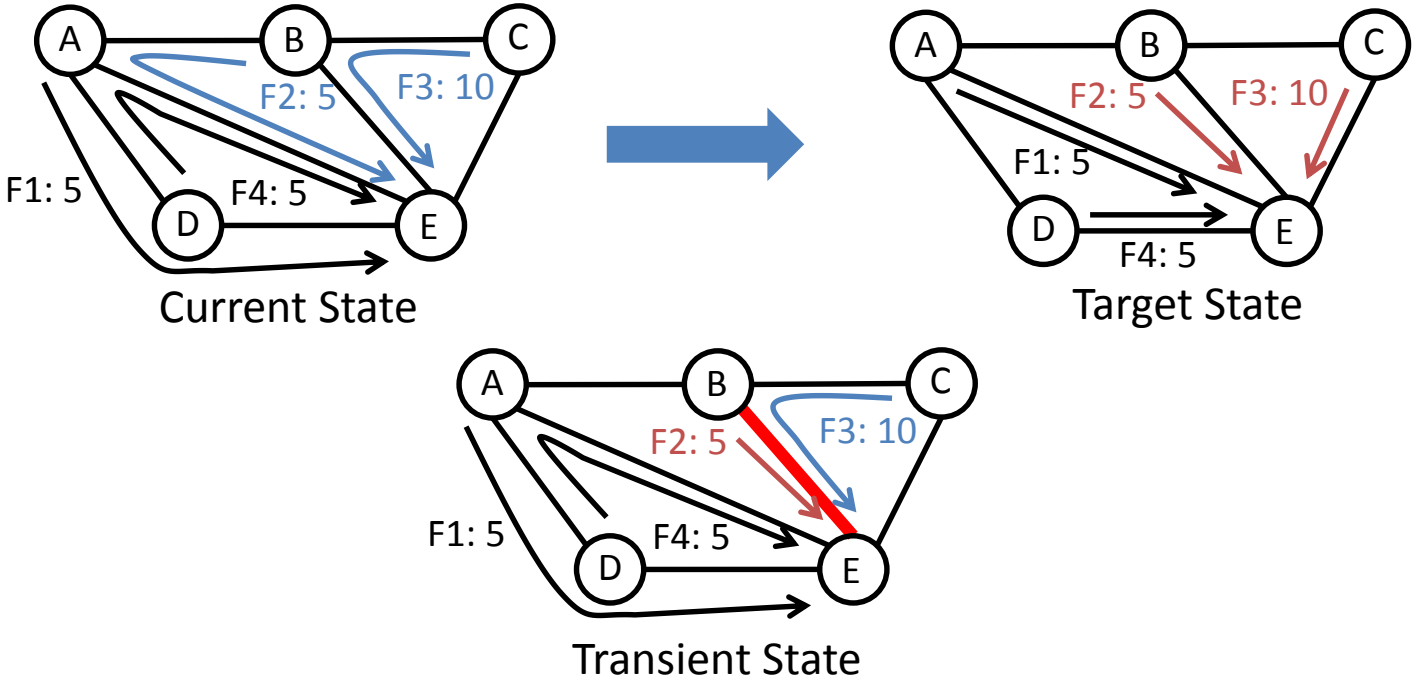
- Requirement 1: fast
  - The agility of control loop
- Requirement 2: consistent
  - No congestion, no blackhole, no loop, etc.



What is Consistent Network Update



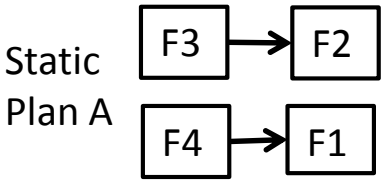
What is Consistent Network Update

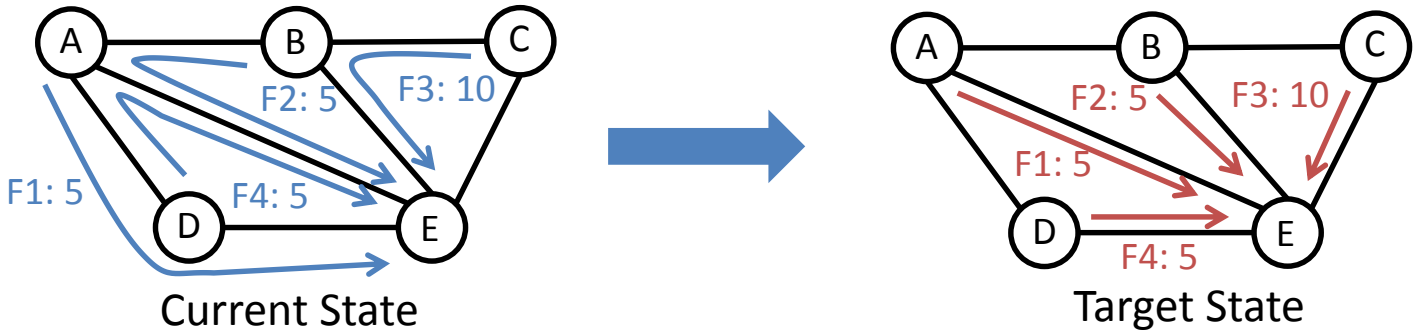


- Asynchronous updates can cause congestion
- Need to carefully order update operations

Existing Solutions are Slow

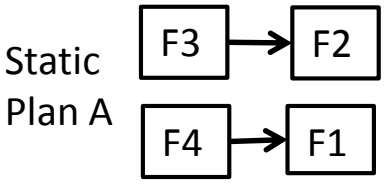
- Existing solutions are static [ConsistentUpdate'12, SWAN'13, zUpdate'13]
  - Pre-compute an order for update operations





Existing Solutions are Slow

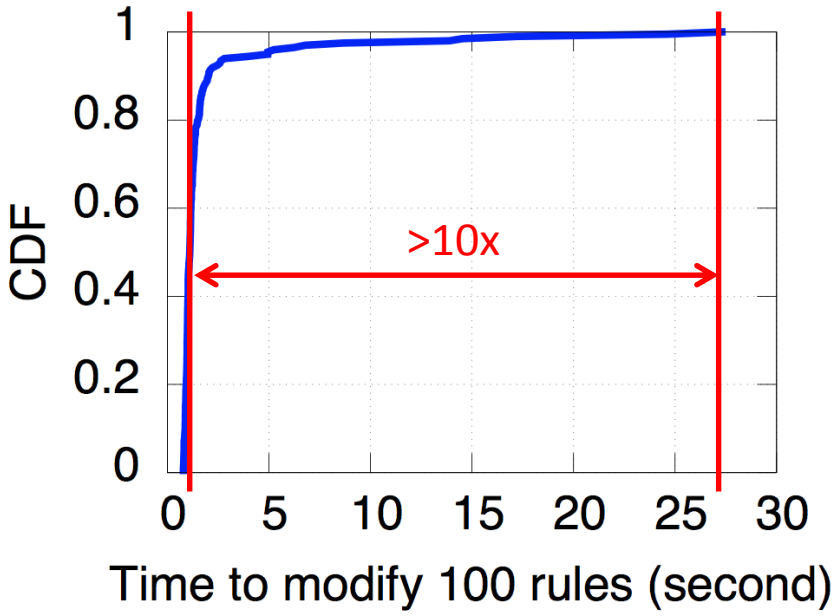
- Existing solutions are static [ConsistentUpdate'12, SWAN'13, zUpdate'13]
  - Pre-compute an order for update operations



- Downside: Do not adapt to runtime conditions
  - Slow in face of highly variable operation completion time

Operation Completion Times are Highly Variable

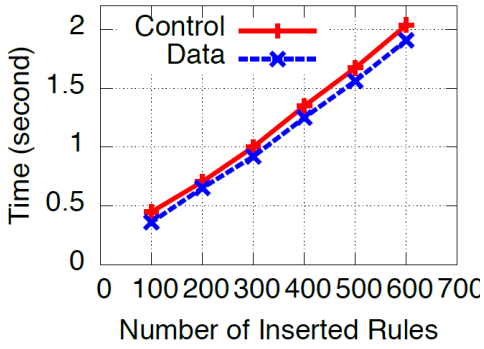
- Measurement on commodity switches

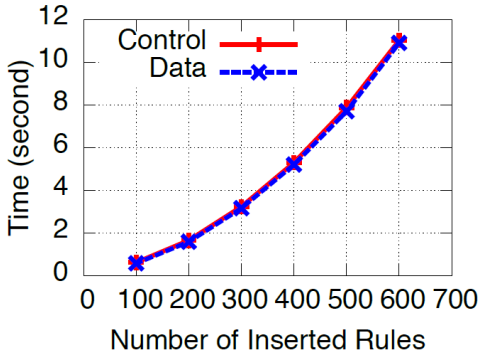


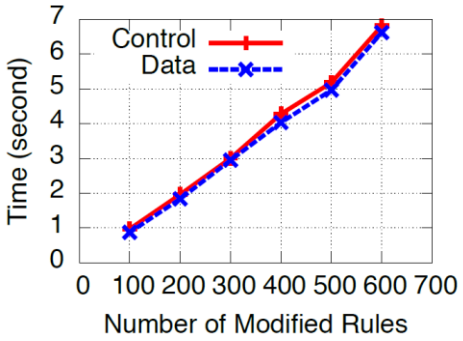
Operation Completion Times are Highly Variable

- Measurement on commodity switches

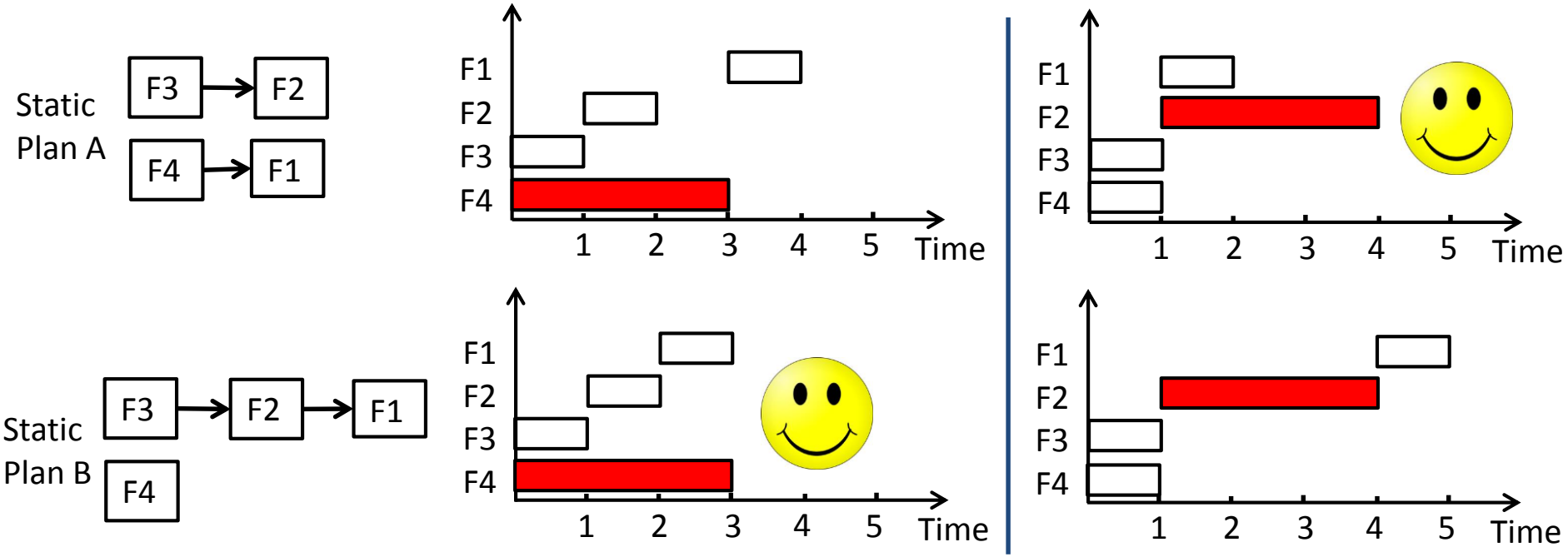
- Contributing factors
  - Control-plane load
  - RPC delays
  - Number of rules
  - Priority of rules
  - Type of operations (insert vs. modify)





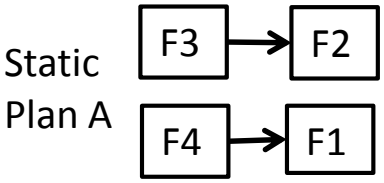


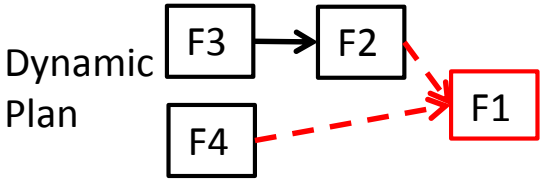
Static Schedules can be Slow



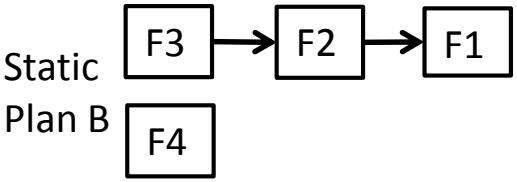
No static schedule is a clear winner under all conditions!

Dynamic Schedules are Adaptive and Fast





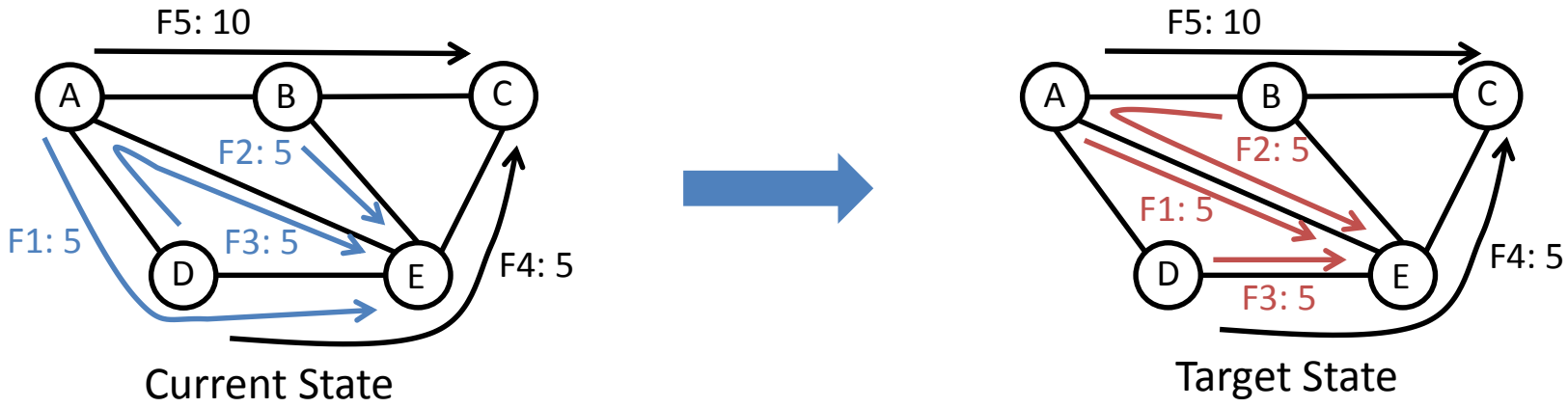
Adapts to actual conditions!



No static schedule is a clear winner under all conditions!

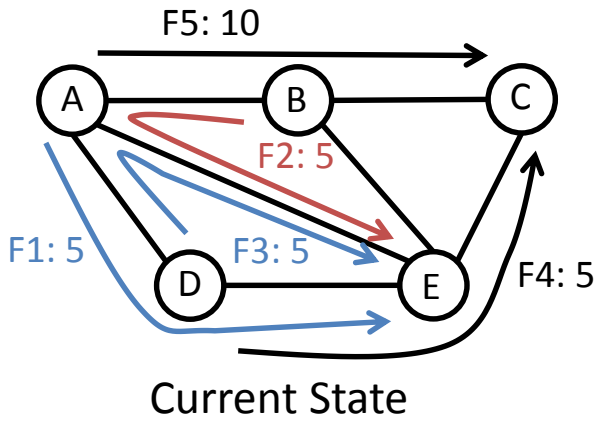
Challenges of Dynamic Update Scheduling

- Exponential number of orderings
  - Integer Linear Programming: slow
- Cannot completely avoid planning
  - Constraint: consistent and feasible



Challenges of Dynamic Update Scheduling

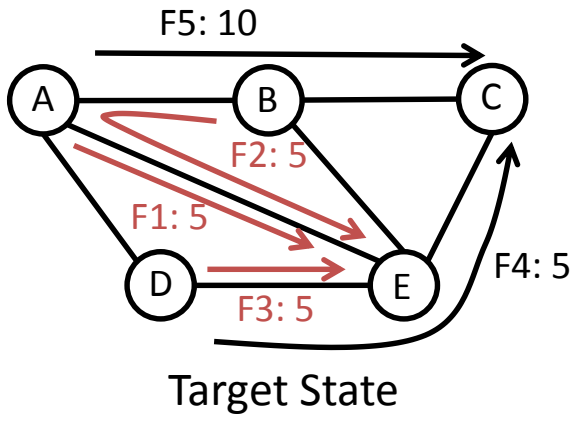
- Exponential number of orderings
  - Integer Linear Programming: slow
- Cannot completely avoid planning
  - Constraint: consistent and feasible



Current State

F2

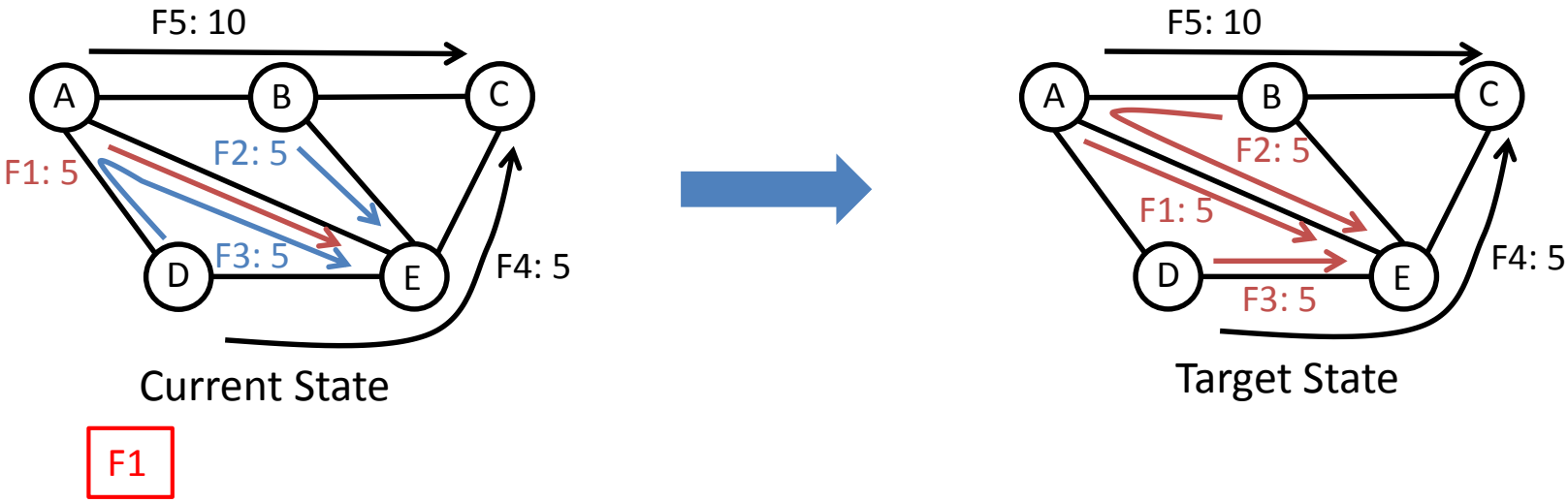
Deadlock



Target State

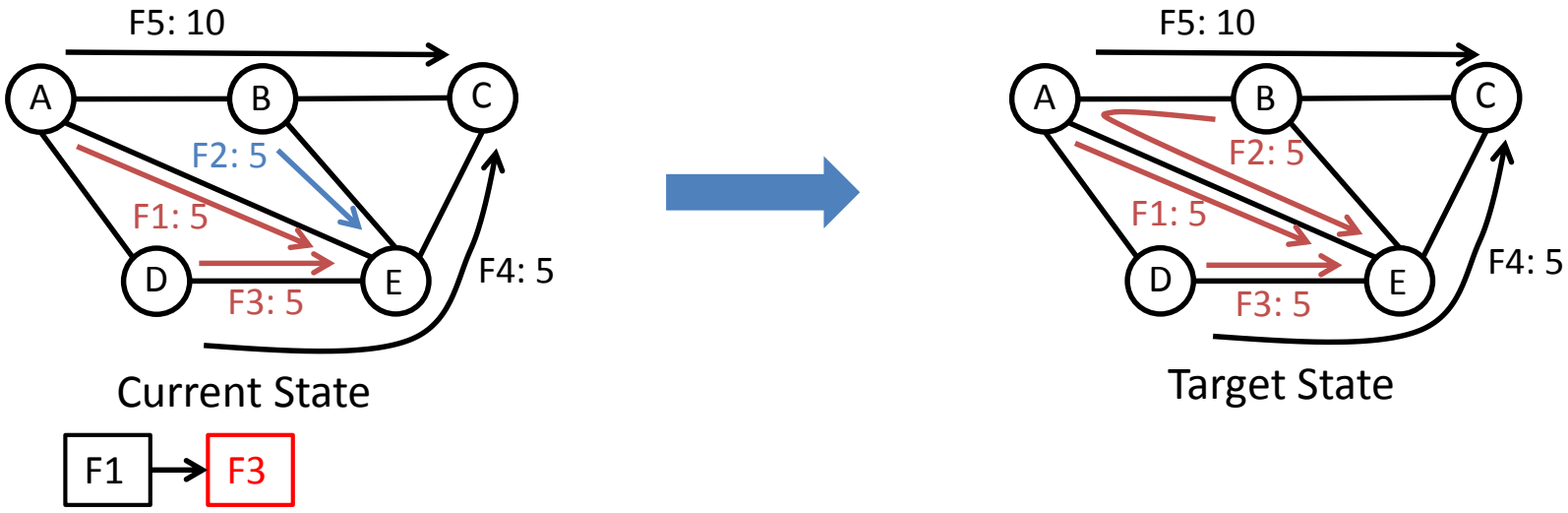
Challenges of Dynamic Update Scheduling

- Exponential number of orderings
  - Integer Linear Programming: slow
- Cannot completely avoid planning
  - Constraint: consistent and feasible



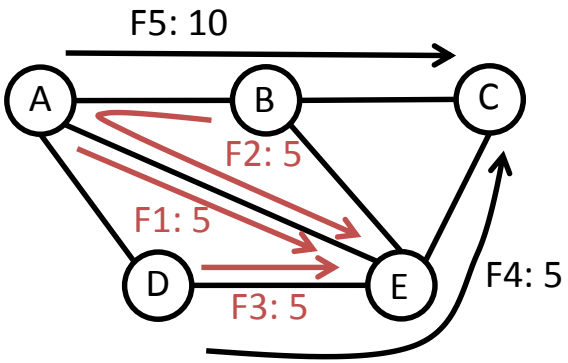
Challenges of Dynamic Update Scheduling

- Exponential number of orderings
  - Integer Linear Programming: slow
- Cannot completely avoid planning
  - Constraint: consistent and feasible



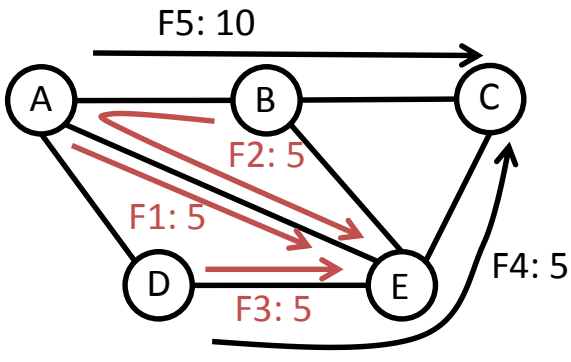
Challenges of Dynamic Update Scheduling

- Exponential number of orderings
  - Integer Linear Programming: slow
- Cannot completely avoid planning
  - Constraint: consistent and feasible



Current State

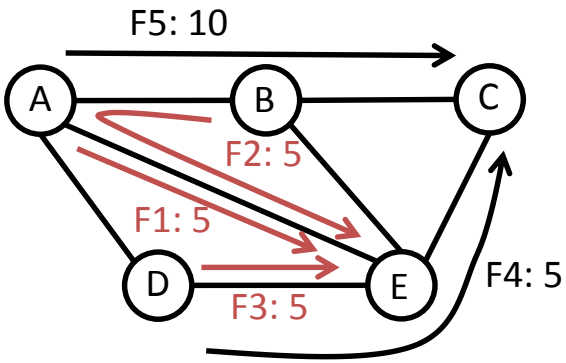
Consistent update plan



Target State

Challenges of Dynamic Update Scheduling

- Exponential number of orderings
  - Integer Linear Programming: slow
- Cannot completely avoid planning
  - Constraint: consistent and feasible

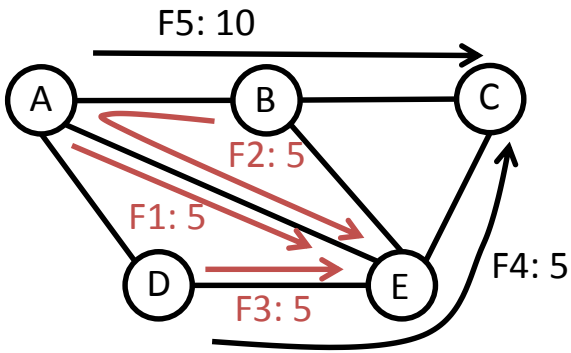


Current State

Consistent update plan

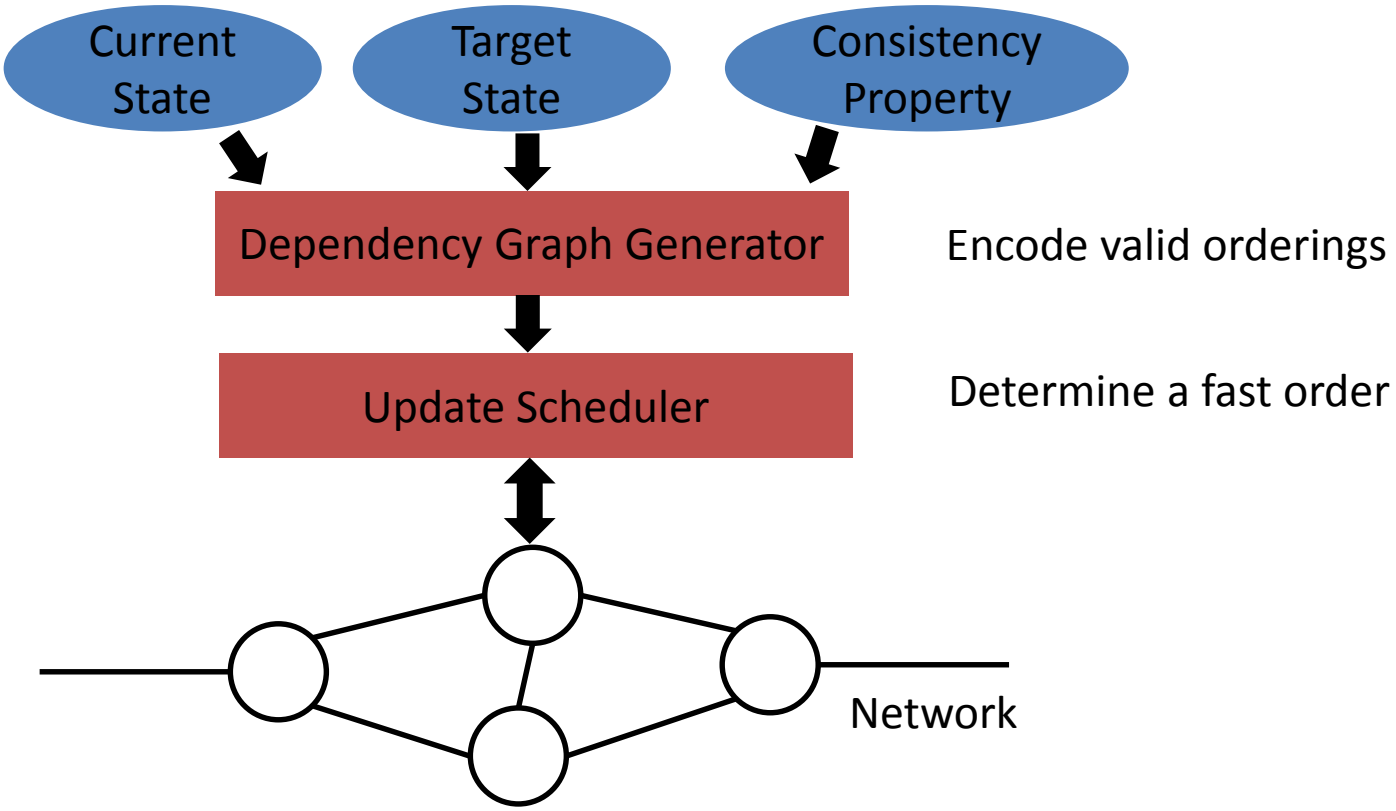
Model

Schedule



Target State

Dionysus Pipeline

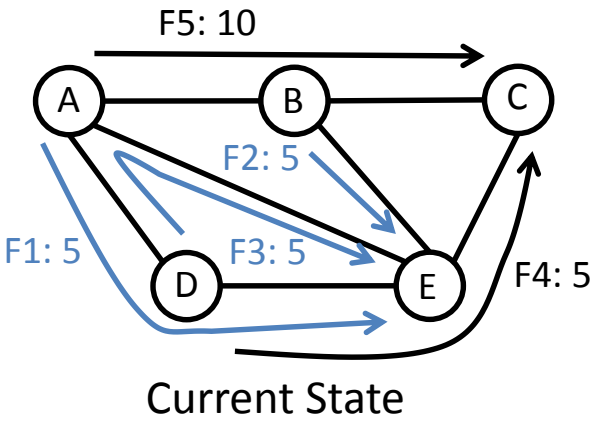


Dependency Graph Generation

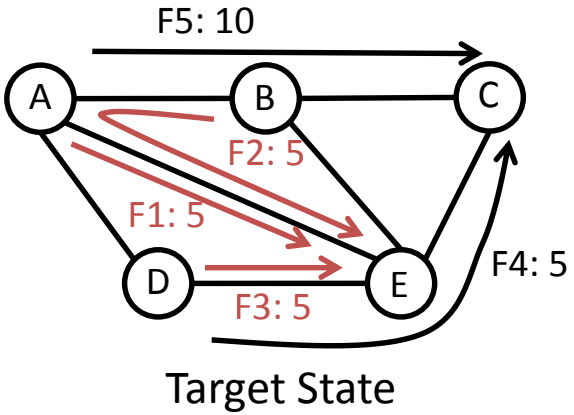
Dependency Graph Elements

Node		Operation node
		Resource node (link capacity, switch table size)
		Path node
Edge		Dependencies between nodes

Dependency Graph Generation



Current State



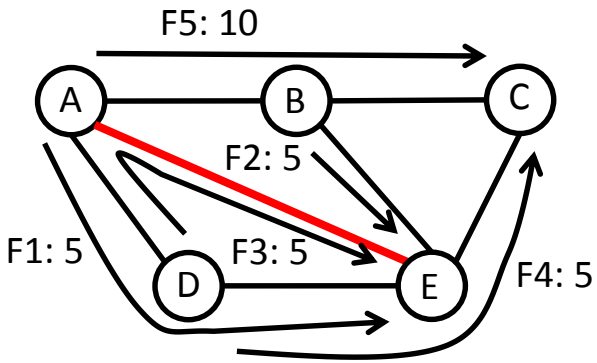
Target State

Move F2

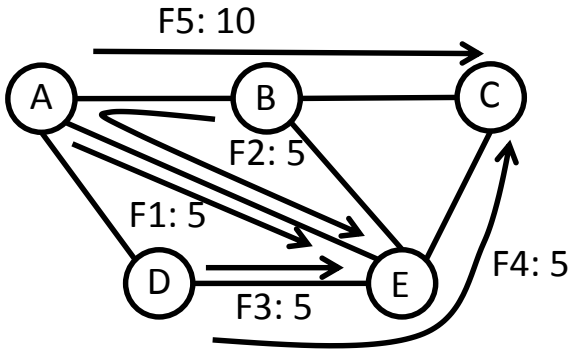
Move F3

Move F1

Dependency Graph Generation



Current State



Target State

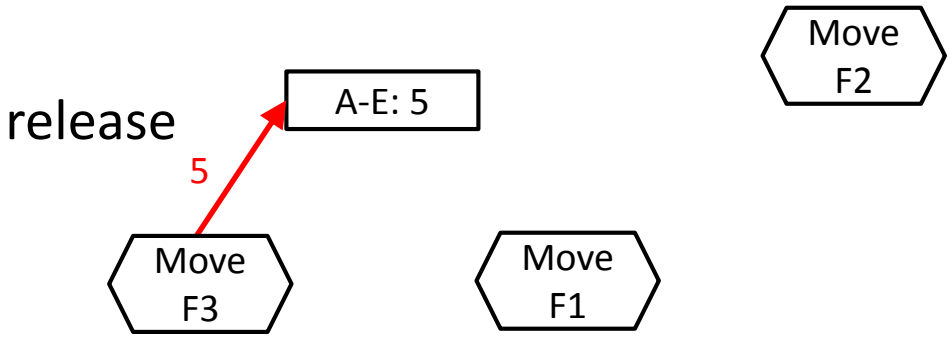
A-E: 5

Move F2

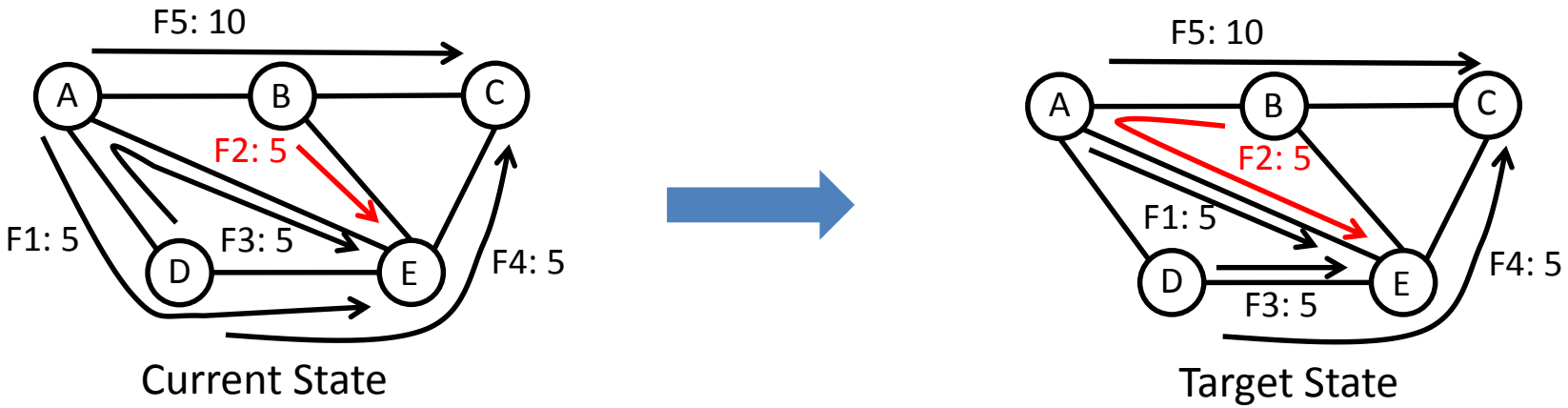
Move F3

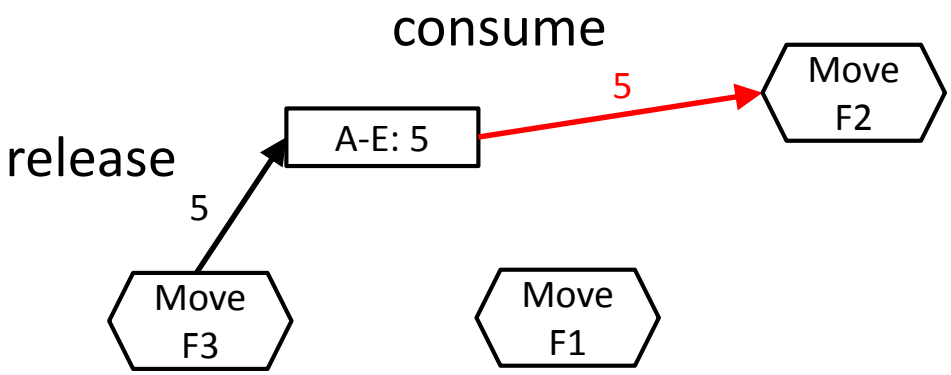
Move F1

Dependency Graph Generation

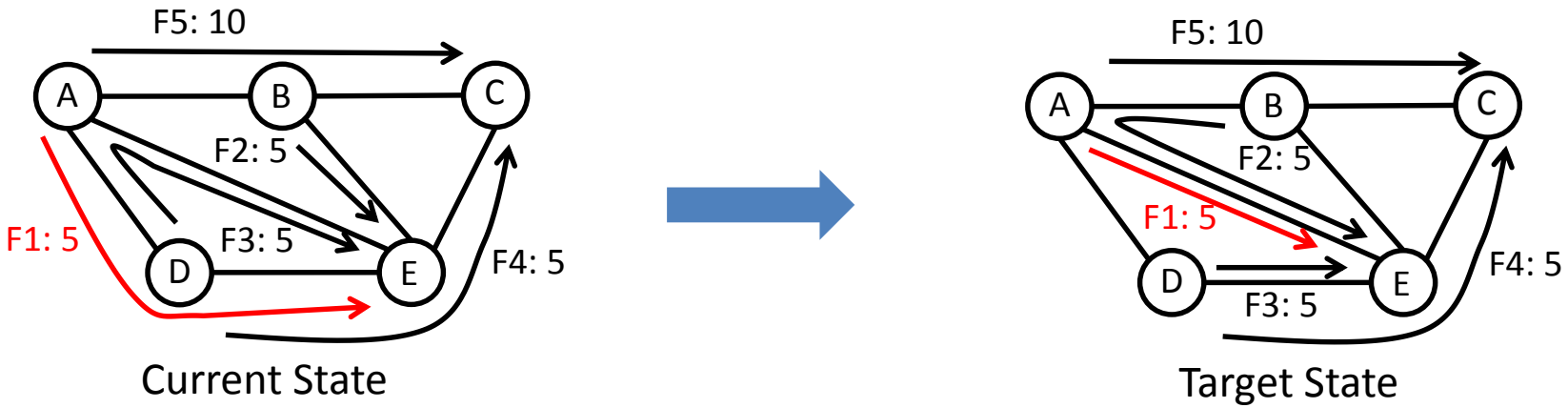


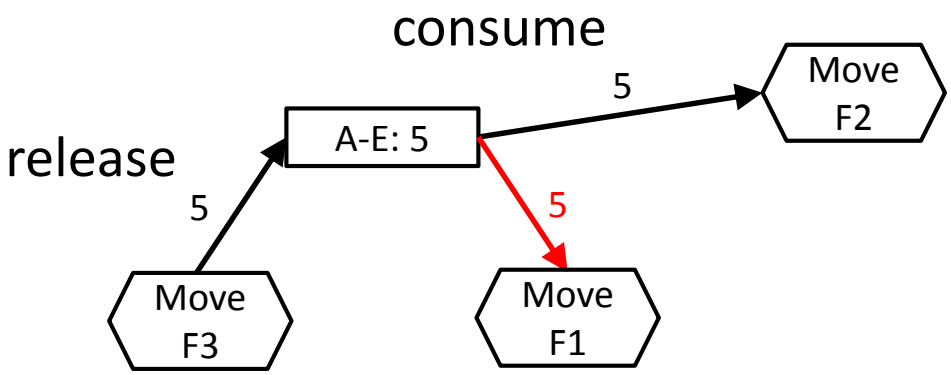
Dependency Graph Generation



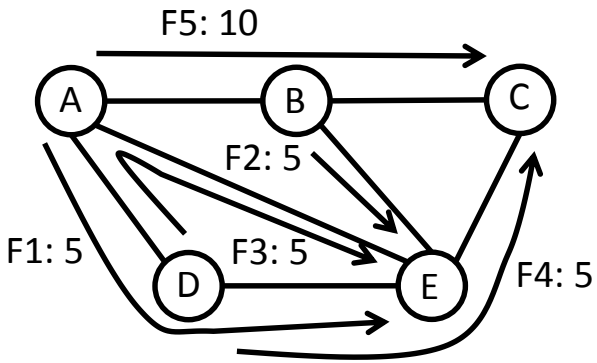


Dependency Graph Generation

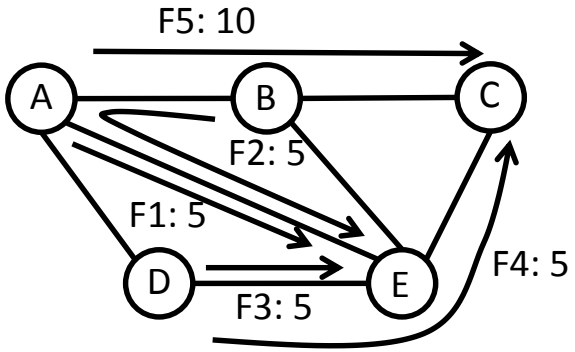




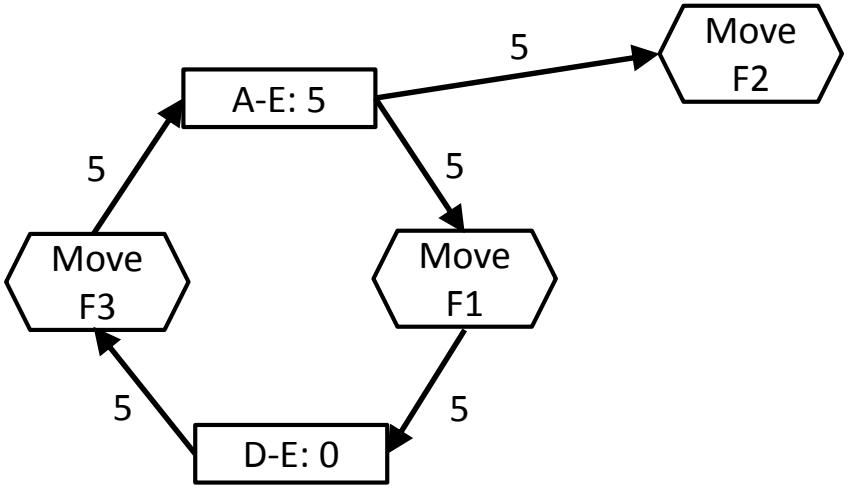
Dependency Graph Generation



Current State



Target State

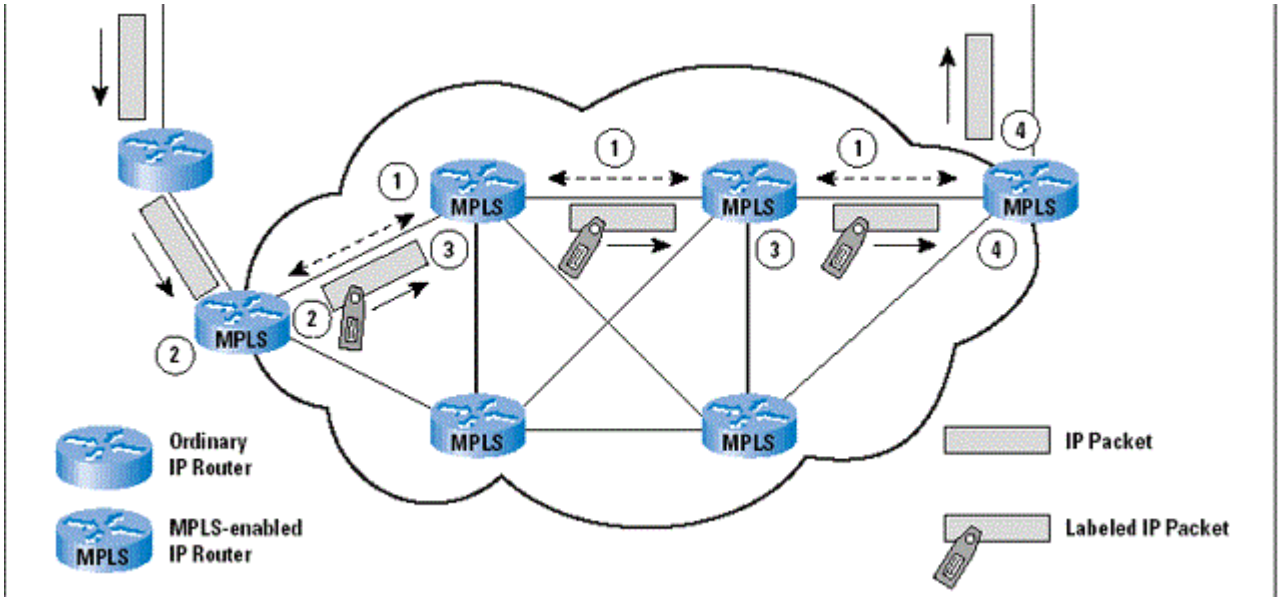


Dependency Graph Generation

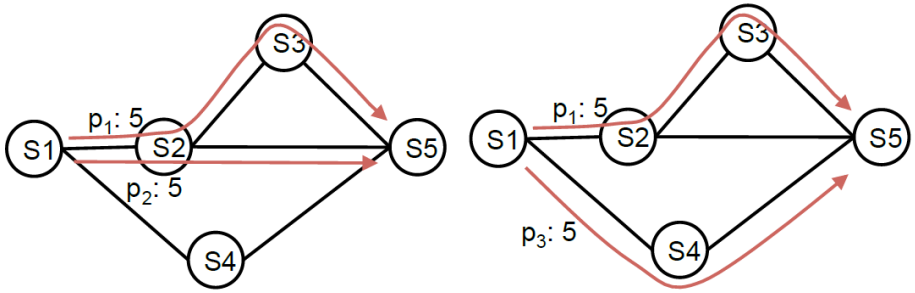
- Supported scenarios
  - Tunnel-based forwarding: WANs
  - WCMP forwarding: data center networks
- Supported consistency properties
  - Loop freedom
  - Blackhole freedom
  - Packet coherence
  - Congestion freedom
- Check paper for details

WAN: Tunnel-based forwarding

- The ingress switches
  - match on packet headers and tag packets with tunnel ids
- Blackhone freedom
  - Tunnels fully established before ingress sws puts any traffic
  - All traffic removed from tunnels before tunnels deleted



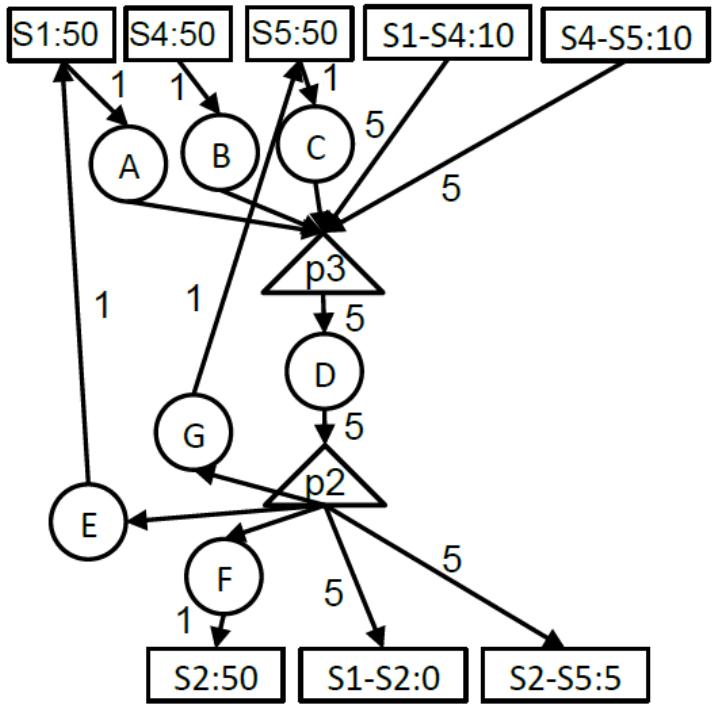
WAN: Tunnel-based forwarding



(a) Current state (b) Target state

Index	Operation
A	Add p_3 at S1
B	Add p_3 at S4
C	Add p_3 at S5
D	Change weight at S1
E	Delete p_2 at S1
F	Delete p_2 at S2
G	Delete p_2 at S5

Table 1: Operations to update f with tunnel-based rules.

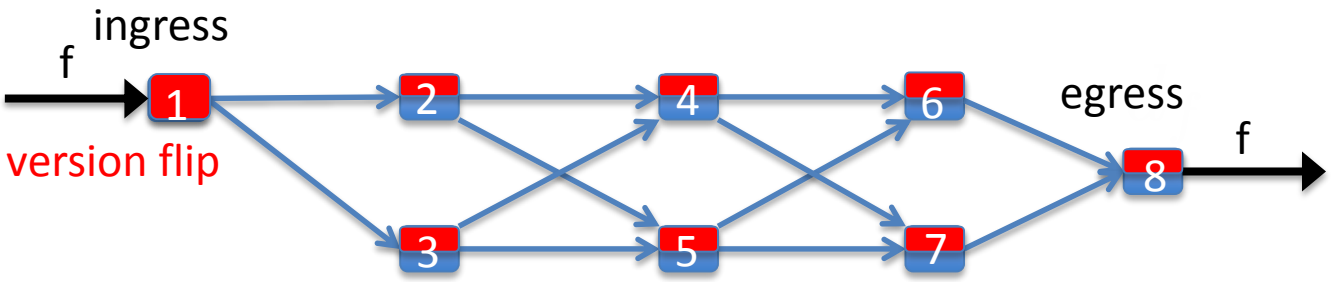


(c) Dependency graph using tunnel-based rules (Table 1)

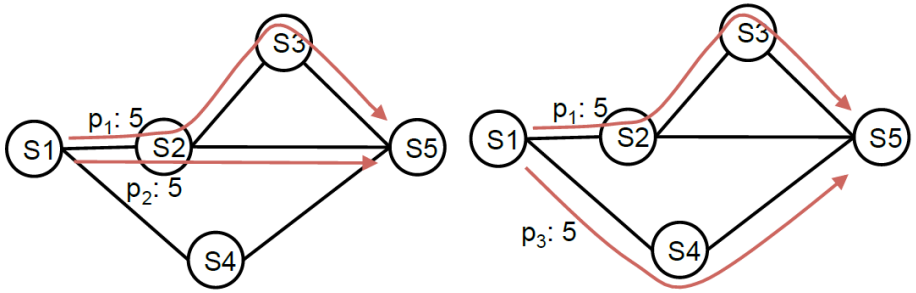
DC: WCMP forwarding

- Switches at every hop
  - match on packet header
- Packet coherence
  - No packet should see a mix of old and new rules
  - Using version numbers

Transition from **old traffic distribution** to **new traffic distribution**



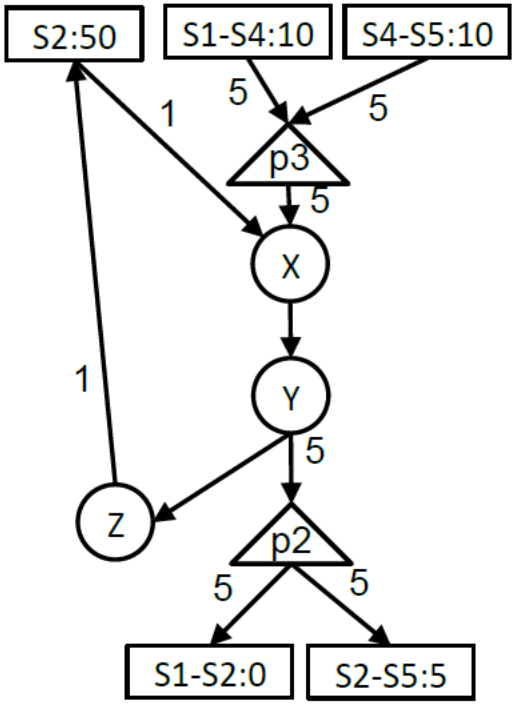
DC: WCMP forwarding



(a) Current state (b) Target state

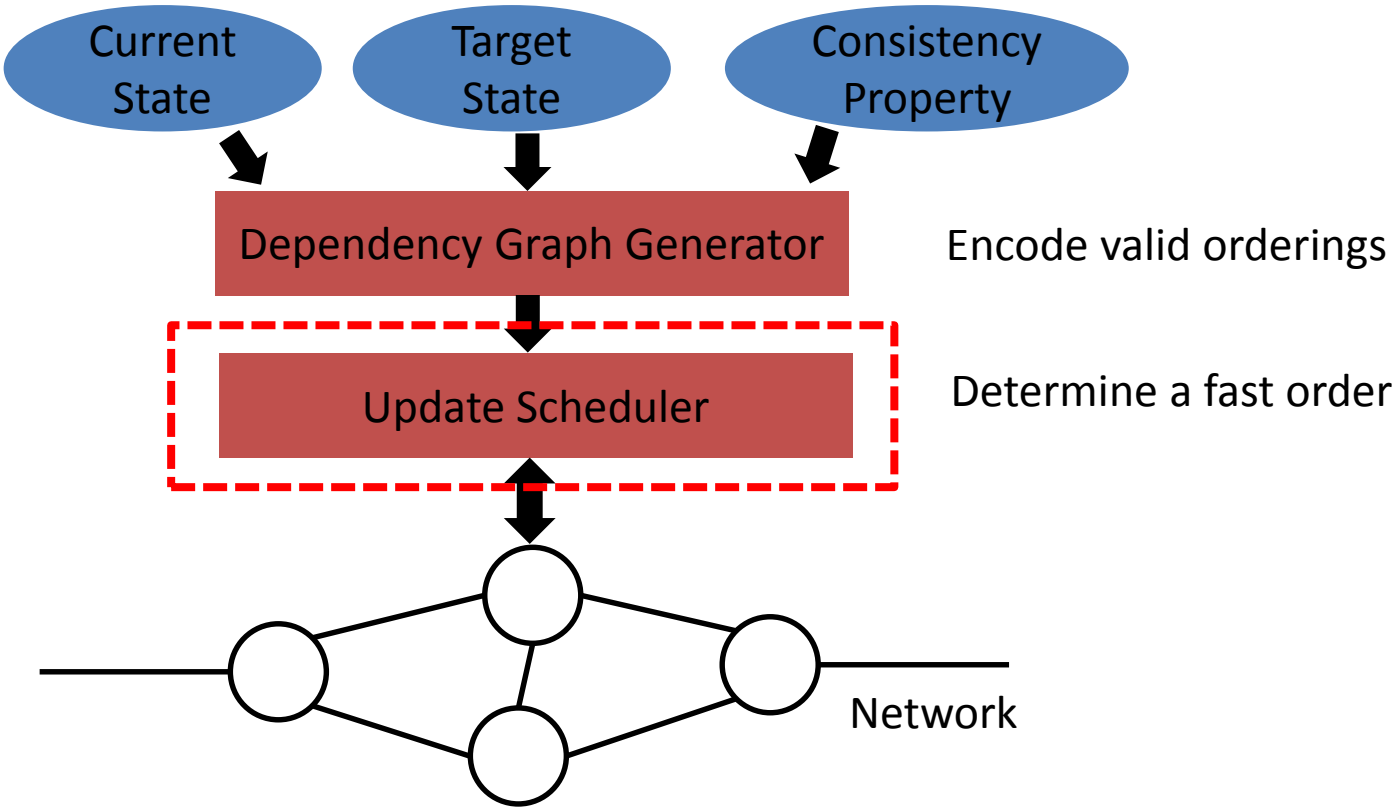
Index	Operation
X	Add weights with new version at S2
Y	Change weights, assign new version at S1
Z	Delete weights with old version at S2

Table 2: Operations to update f in WCMP forwarding.



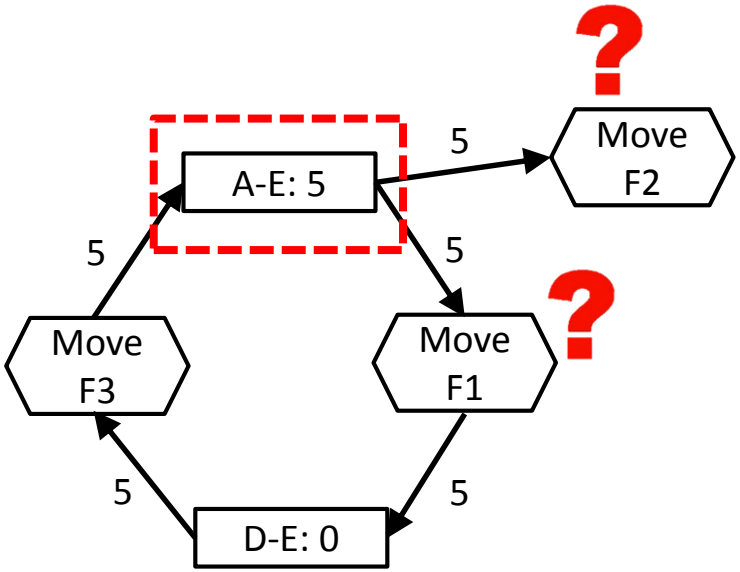
(d) Dependency graph using WCMP-based rules (Table 2)

Dionysus Pipeline



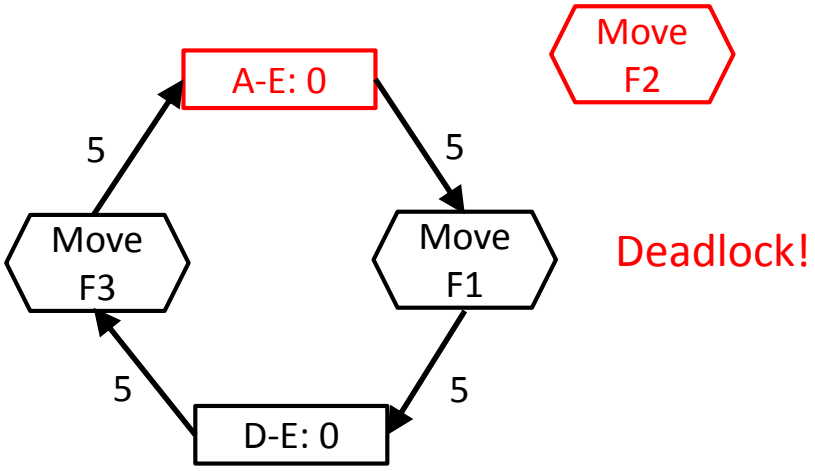
Dionysus Scheduling

- Scheduling as a resource allocation problem



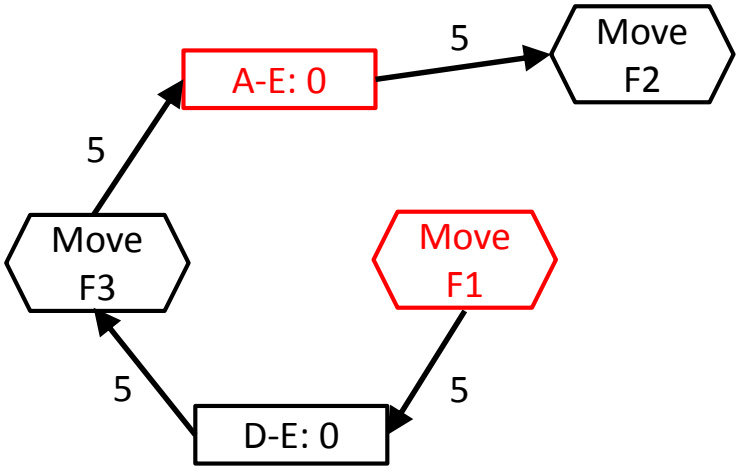
Dionysus Scheduling

- Scheduling as a resource allocation problem



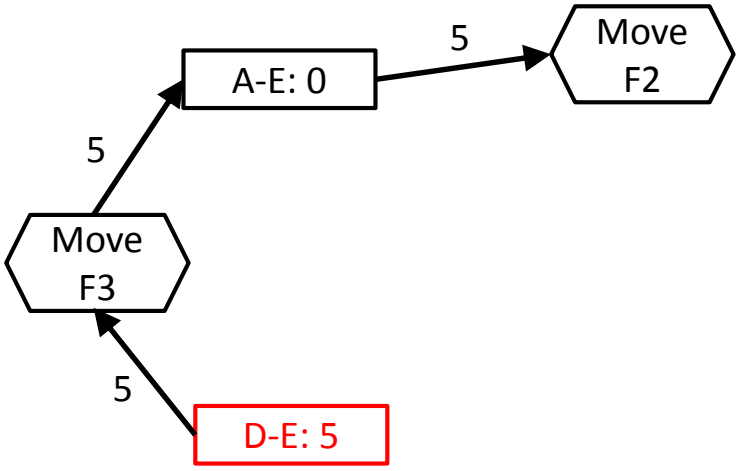
Dionysus Scheduling

- Scheduling as a resource allocation problem



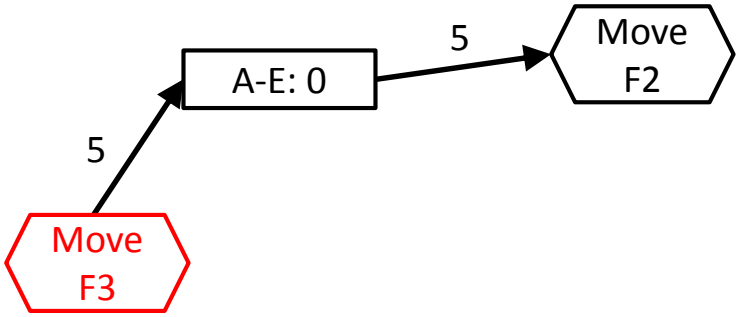
Dionysus Scheduling

- Scheduling as a resource allocation problem



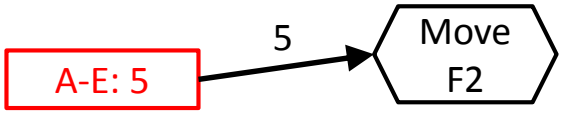
Dionysus Scheduling

- Scheduling as a resource allocation problem



Dionysus Scheduling

- Scheduling as a resource allocation problem



Dionysus Scheduling

- Scheduling as a resource allocation problem

Dionysus Scheduling

- Scheduling as a resource allocation problem
- Finding a feasible update is NP-complete under link capacity and switch table size constraints
- Approach
  - Directed acyclic graph (DAG): always feasible, critical-path
  - General case: covert to a virtual DAG
  - Rate limit flows to resolve deadlocks

Critical-Path Scheduling

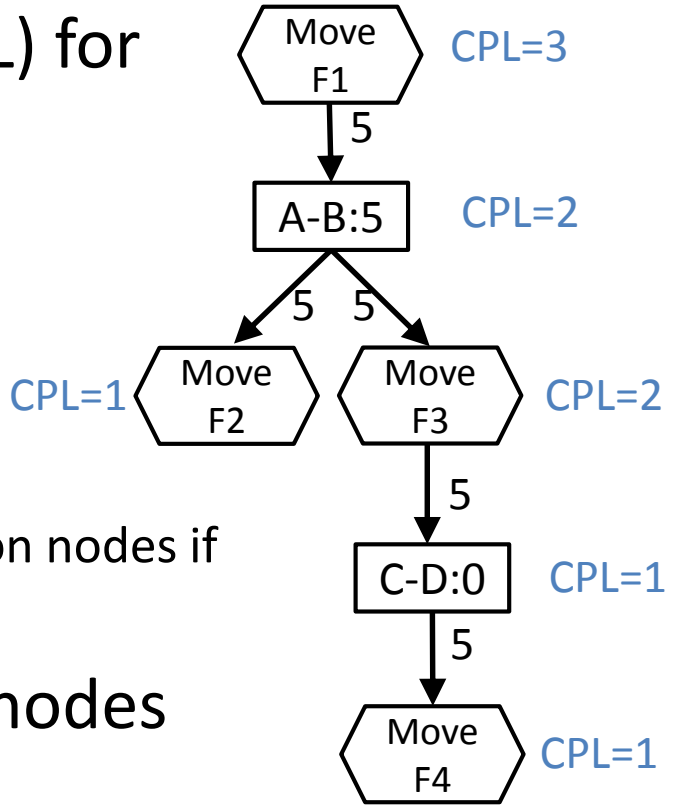
- Calculate critical-path length (CPL) for each node

$$CPL_i = w_i + \max_{j \in children(i)} CPL_j$$

$$w_i = \begin{cases} 1, if\ i\ is\ operation\ node \\ 0, otherwise \end{cases}$$

  - Extension: assign larger weight to operation nodes if we know in advance the switch is slow

- Resource allocated to operation nodes with larger CPLs



Critical-Path Scheduling

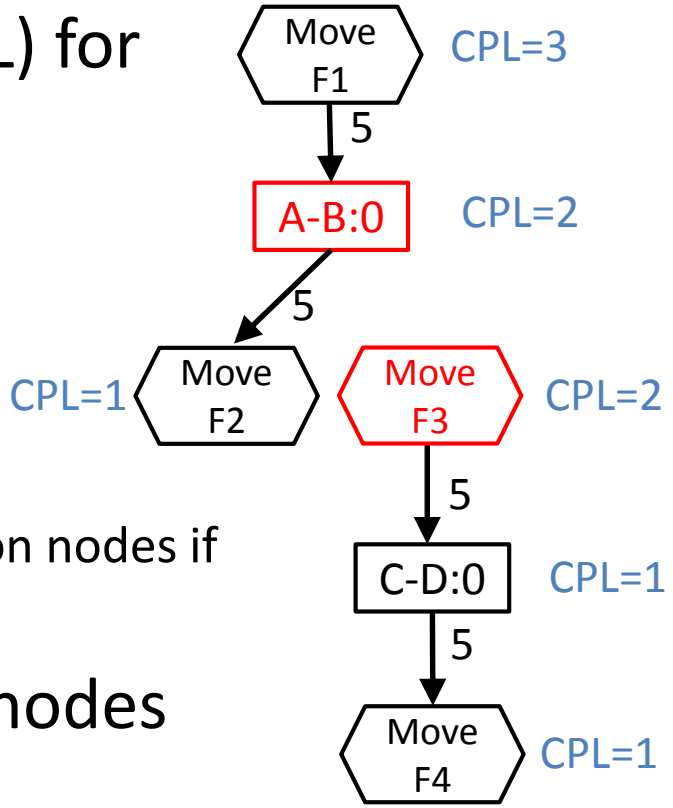
- Calculate critical-path length (CPL) for each node

$$CPL_i = w_i + \max_{j \in children(i)} CPL_j$$

$$w_i = \begin{cases} 1, if\ i\ is\ operation\ node \\ 0, otherwise \end{cases}$$

  - Extension: assign larger weight to operation nodes if we know in advance the switch is slow

- Resource allocated to operation nodes with larger CPLs

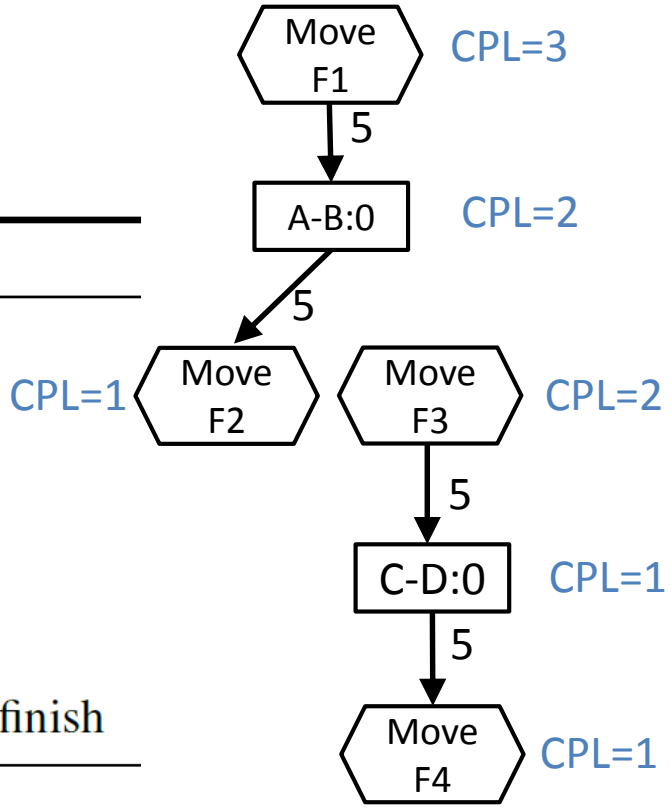


Scheduling DAGs

- Dynamically schedule

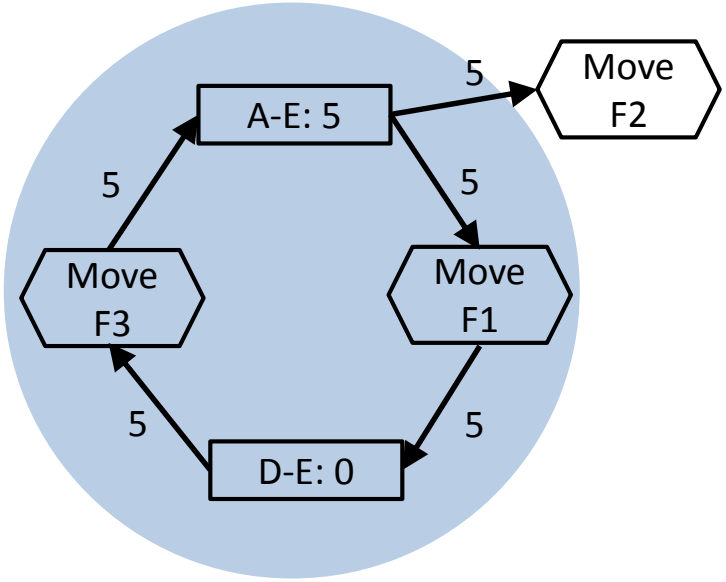
Algorithm 2 ScheduleGraph(G)

```
1: while true do
2:     UpdateGraph(G)
3:     Calculate CPL for every node
4:     Sort nodes by CPL in decreasing order
5:     for unscheduled operation node O_i ∈ G do
6:         if CanScheduleOperation(O_i) then
7:             Schedule O_i
8:     Wait for time t or for all scheduled operations to finish
```

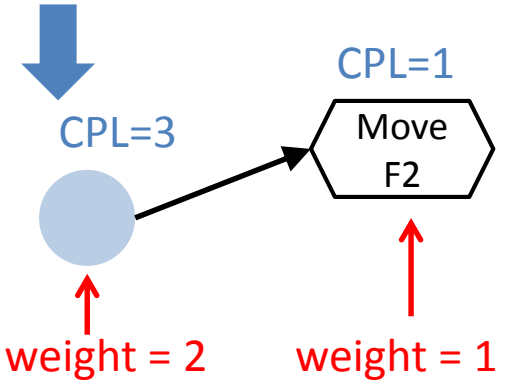


Handling Cycles

- Convert to virtual DAG
  - Consider each strongly connected component (SCC) as a virtual node

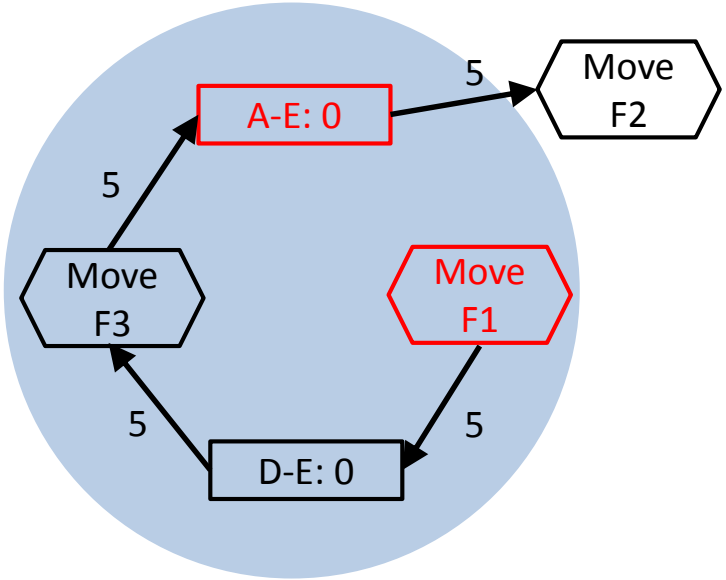


- Critical-path scheduling on virtual DAG
  - Weight w_i of SCC: number of operation nodes

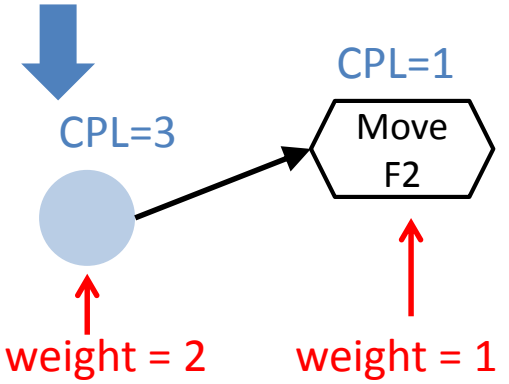


Handling Cycles

- Convert to virtual DAG
  - Consider each strongly connected component (SCC) as a virtual node



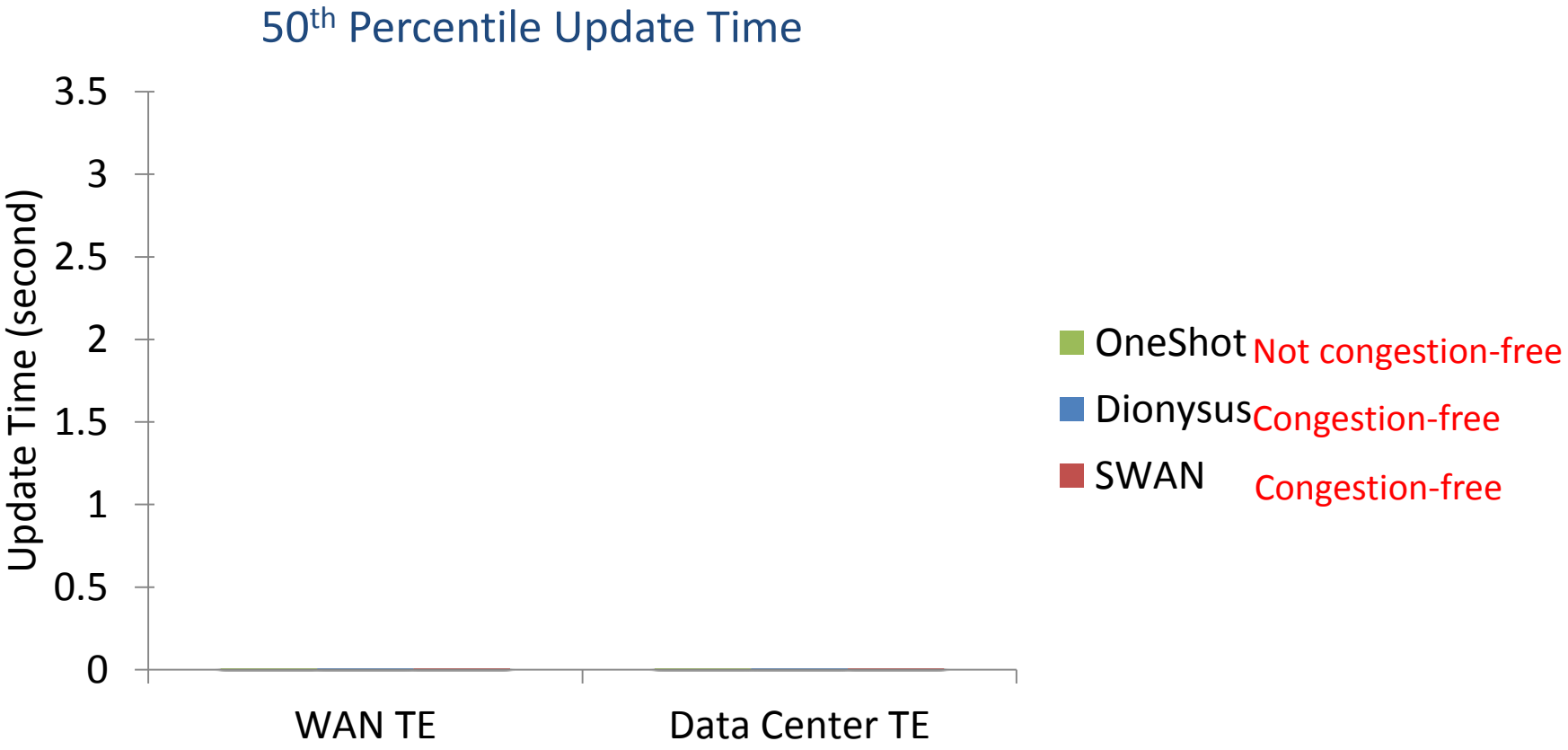
- Critical-path scheduling on virtual DAG
  - Weight w_i of SCC: number of operation nodes



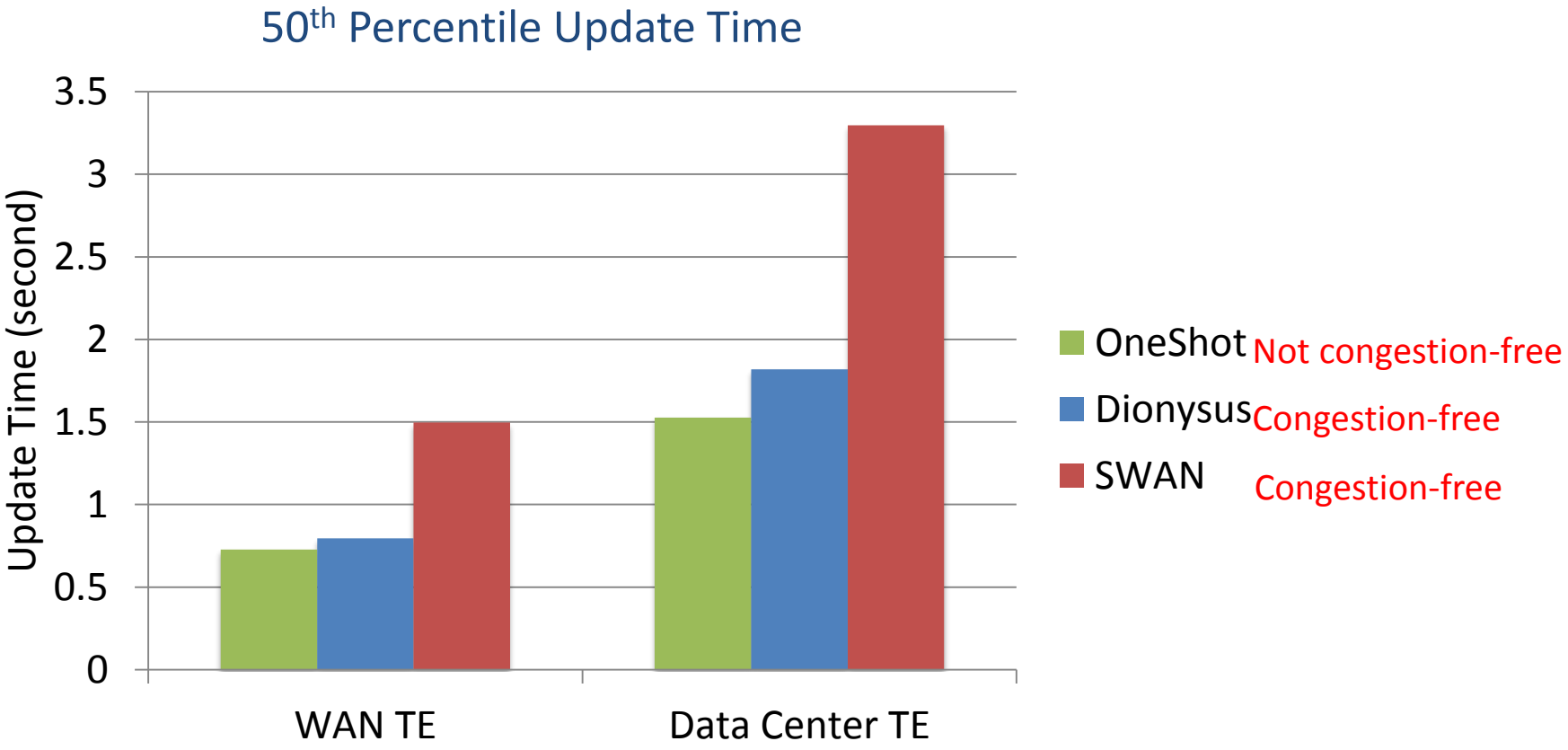
Evaluation

- Implementation
  - Prototype: 5,000+ lines of C# code
- Datasets
  - WAN
    - interconnects $O(50)$ sites
    - 288 traffic matrices on a day
  - DC
    - 3 layers with several hundred switches

Evaluation: Traffic Engineering



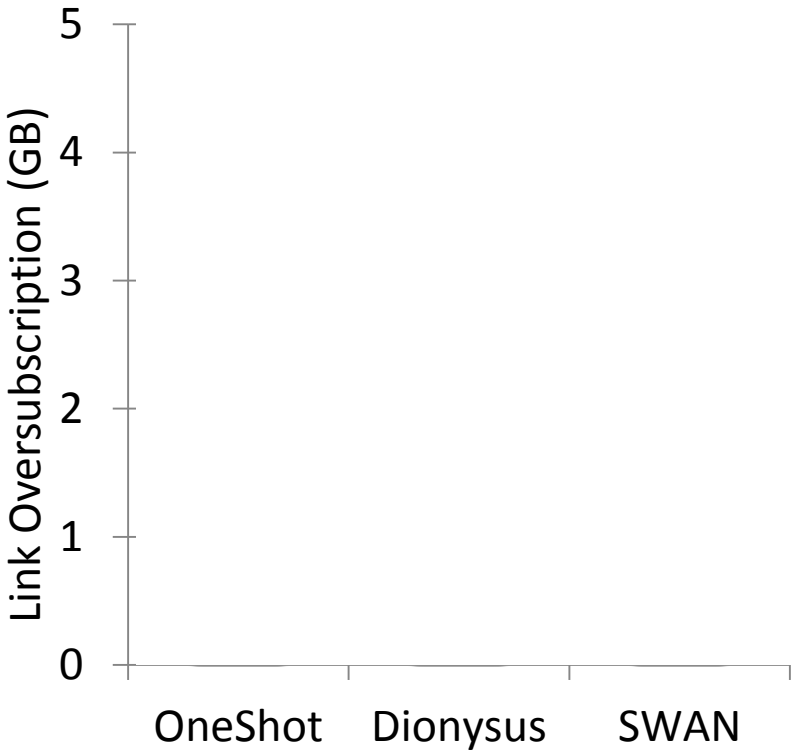
Evaluation: Traffic Engineering



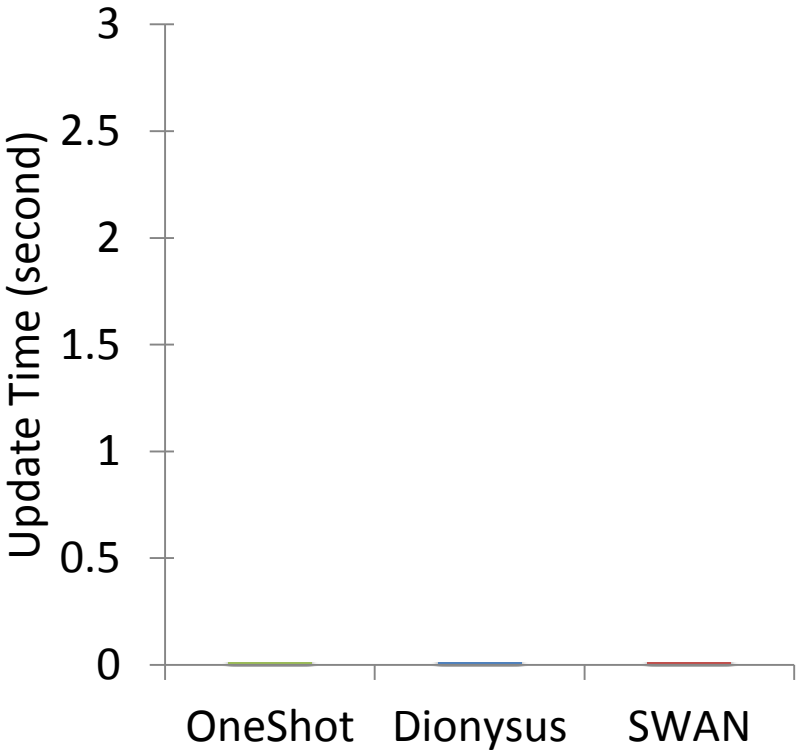
Improve 50th percentile update speed by 80%
compared to static scheduling (SWAN), close to OneShot

Evaluation: Failure Recovery

99th Percentile
Link Oversubscription

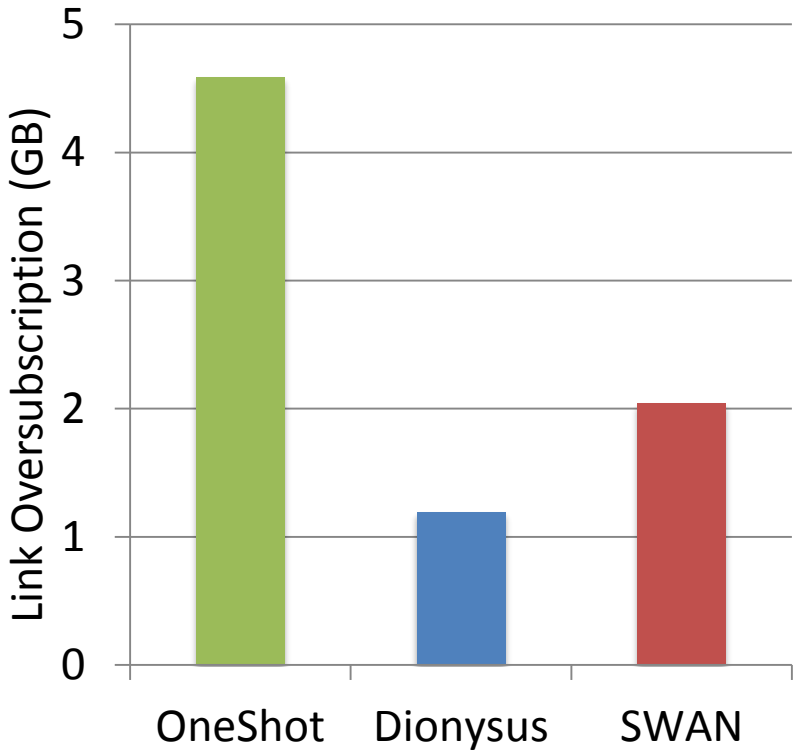


99th Percentile
Update Time

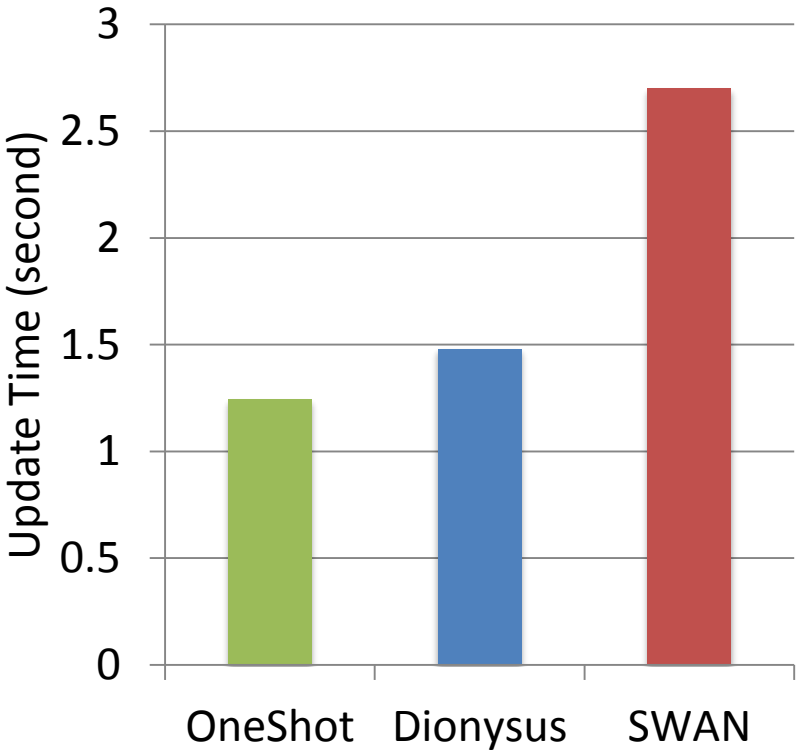


Evaluation: Failure Recovery

99th Percentile
Link Oversubscription



99th Percentile
Update Time



Reduce 99th percentile link oversubscription by 40% compared to static scheduling (SWAN)

Improve 99th percentile update speed by 80% compared to static scheduling (SWAN)

Conclusion

- Dionysus provides fast, consistent network updates through dynamic scheduling
  - Dependency graph: compactly encode orderings
  - Scheduling: dynamically schedule operations

Dionysus enables more agile SDN control loops

Thanks!