Network Updates for the Impatient: Eliminating Unnecessary Waits

Hossein Hojjat
Cornell University

Jedidiah McClurg
CU Boulder

Pavol Černý
CU Boulder

Nate Foster
Cornell University

Introduction. Identifying the correct sequence of management operations to apply during a network update is a challenging task that can easily go wrong when performed manually [2]. Ad hoc update strategies can lead to a host of anomalies including dropped connections, forwarding loops, access control violations, and more. In recent work, Reitblatt et al. proposed the notion of a consistent network update [5], which guarantees that packets are processed by a single policy all the way through the network. Unfortunately, consistent updates can require doubling the amount of memory on switches in general—something that is not practical in large networks since switches rely on expensive and power-hungry ternary content-addressable memories. A different approach, recently proposed by McClurg et al. is to use program synthesis to generate updates that preserve invariants specified by the programmer [4]. The idea is to explore the space of possible orderings of updates to individual switches, searching for a sequence that reaches the target configuration while ensuring that the invariants are not violated even for the intermediate configurations. After some (though not all) updates to individual switches, the synthesized program needs to wait for all in-flight packets to exit the network (see the example below). For simplicity and efficiency of synthesis, the algorithm in [4] waits after each individual switch update. A heuristic is then used to eliminate spurious waits (e.g. between switches in disjoint parts of a network) but there is no guarantee that the number of waits in the synthesized update will be optimal.

Contributions. This paper develops a novel relationship between networks and concurrent programs, and leverages this connection to reduce the problem of *minimizing the number of waits required for a network update* to the problem of *inserting the minimal number of fences in a program running on a weak memory model* [1]. We focus on the PSO (Partial Store Ordering) weak memory model, which allows store operations to disjoint locations to be reordered with respect to earlier store operations performed by the same thread. PSO is attractive because its safety verification problem is decidable and there exist tools for finding minimal fences [3]. We present a faithful translation from networks to programs and exploit algorithms for synthesizing minimal fences to identify the necessary waits. We have implemented a simple prototype that uses Linden and Wolper's tool [3] on simple benchmarks and confirmed that it finds the minimal number of waits for a number of simple updates provided as inputs.

Example. Figure 1 depicts a simple network before and after an update. In this network the ingress switch s_1 filters SSH traffic going to s_2 and forwards all traffic to s_3. To better balance the traffic the network administrator wants to configure the configuration of the network by performing three updates: upd_1 (enable link s_1-LAN_3), upd_2 (disable link s_2-LAN_1 and enable link s_2-LAN_2), and upd_3 (disable link s_3-s_2). However, throughout the update, we wish to ensure that at all times the network prevents SSH traffic from being sent to LAN_2. A possible update sequence is upd_3;upd_2;upd_1. In order to preserve the required property, we need to wait after upd_3. The reason is that if we perform upd_2 right away, there might be some in-flight packets that were processed by s_3 before upd_3, and by s_2 after upd_2. These might be SSH packets that enter LAN_2.

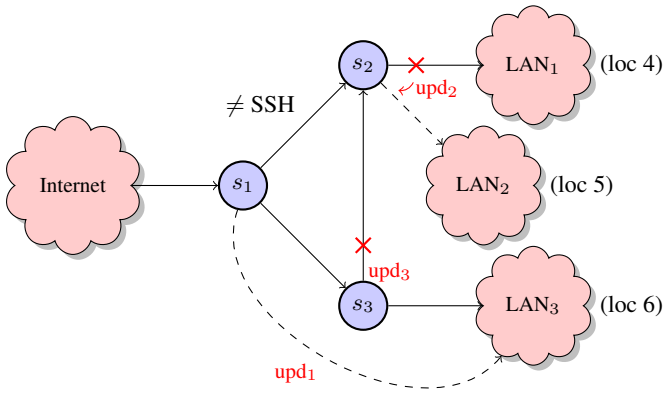
From Networks to Concurrent Programs. To analyze updates using standard tools, we define a translation that maps a network and an update to a concurrent program with three threads: DataPlane, ControlPlane and Assert. Together, these threads model the behavior of an individual packet traversing the network during the update. We assume that switches and hosts have unique integer identifiers (e.g., 1, 2, 3, etc.) as shown in Figure 1. The program maintains global variables u_{sw} and h_{sw} for each switch sw, as well as a collection of variables $w_{\mathcal{I}}$ in which $\mathcal{I}$ is a subset of the switches. The u_{sw} variable tracks whether switch sw has been updated: it is set to 0 initially, and is updated to 1 when the ControlPlane decides to update sw. The h_{sw} variable tracks whether the packet passes through in its initial configuration, final configuration, or not at all: it is set to 0 initially, and is set to 1 if the packet is processed by sw in the initial configuration and 2 if it is processed by sw in the final configuration. We assume that the network is loop-free, so packets are processed by each switch at most once. The wait variables $w_{\mathcal{I}}$ keep track of a set of switches $\mathcal{I}$ that have been updated. The DataPlane thread uses wait variables to rule out certain "impossible" executions: if $w_{\mathcal{I}}$ is set to 1, then the packet should have only seen the final configurations of the updated switches (as tracked by the h_{sw} variables). The program prevents the packet from seeing the initial configurations of those switches by assuming that the value of the corresponding h_{sw} variables is not 1. The rest of the DataPlane thread uses a set of atomic conditional statements to model a single hop of processing on a switch. The Assert thread simply encodes the desired network invariant ϕ.

Weak Memory Models. For static networks, where the initial and final configurations are identical, it is straightforward to show that the network and the translated program produce the same set of traces under any memory model:

THEOREM 1 (Static Trace Equivalence). *Let N be a network with an empty sequence of update commands and let the program Π be the corresponding static network program. For both sequentially consistent and PSO memory models, $N \simeq \Pi$.*

However, in the presence of non-trivial updates, the situation is different. We can prove a similar simulation result for dynamic networks for a sequentially consistent memory:

THEOREM 2 (SC Trace Equivalence). *Let N be a network with a sequence of update commands and a number of waits at different steps and let Π be the corresponding network program. If we execute Π on a sequentially consistent memory model, $N \simeq \Pi$.*



$u_1 = 0, u_2 = 0, u_3 = 0$
$h_1 = 0, h_2 = 0, h_3 = 0$
$w_{\{3\}} = 0, w_{\{2,3\}} = 0,$
$loc = 1$
$0 \leq type \leq 1$

Process DATAPLANE
1: **loop**
2: **if** $(loc = 2)$ **then**
3: **atomic**
4: **if** $(w_{\{3\}} = 1)$ **then assume** $(h_3 \neq 1)$
5: **if** $(w_{\{2,3\}} = 1)$ **then assume** $(h_2 \neq 1 \wedge h_3 \neq 1)$
6: **if** $(u_2 = 0)$ **then** $(loc \leftarrow 4)$; $h_2 \leftarrow 1$
7: **if** $(u_2 = 1)$ **then** $(loc \leftarrow 5)$; $h_2 \leftarrow 2$
8: **else**
9: /* Similar to case for loc = 2 */

Process CONTROLPLANE
10: **atomic** $(u_3 \leftarrow 1$; $w_{\{3\}} \leftarrow 1)$;
11: **atomic** $(u_2 \leftarrow 1$; $w_{\{2,3\}} \leftarrow 1)$;
12: $u_1 \leftarrow 1$

Process ASSERT
13: **assert**$\neg($ $(loc = 5) \wedge (type = 0)$ $)$

Figure 1. Example update and translated program.

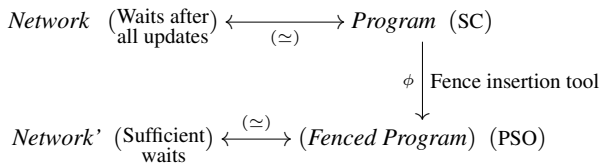
Unfortunately, the analogous theorem does not hold under the PSO weak memory model, since the assignments generated by the control plane may be committed to memory in any order. Among other issues, PSO may potentially reorder the assignments to update variables and postpone assignments to wait variables. In extreme cases, all the assignments to the wait variables may be postponed until the end of the execution of the control plane process—i.e., after all the switches have been updated.

Fence Synthesis and Minimal Waits. To prevent the undesired behavior of swapping the assignments, PSO memory models typically provide a fence instruction that enforces a boundary between assignments occurring before and after the fence. A conservative way to restore the sequentially-consistent execution is to insert a fence between every assignment, although this leads to less efficient code. Fortunately, there exist fence synthesis tools which attempt to insert only those fences needed to ensure the required correctness property. When applied to the program corresponding to a network, such a tool inserts fences after assignments to the waits needed to ensure the property encoded in the Assert thread. The following theorem captures this relationship formally:

THEOREM 3 (PSO Trace Equivalence). *Let N be a network with a sequence of update commands and a number of waits at different steps and let Π be the corresponding fenced network program. If we execute Π on a PSO weak memory model, $N \simeq \Pi$.*

As an example, given the program shown in Figure 1, a fence synthesis tool would place a single wait after the line 10. This fence conveys that the updates after the fence (1 and 2) can be done in parallel, but we must wait after updating 3 before continuing to update the remaining switches. This fence corresponds to the actual waits that would be needed in the network.

Summary. The following diagram depicts the overall algorithm for removing waits using a fence synthesis tool:



First, we use an update synthesis tool to generate a correct update sequence with a wait between each individual switch update. Next, we translate the network to an equivalent concurrent program. Finally, we use a fence synthesis tool to identify the fences needed to ensure correctness and read off the corresponding waits. The three theorems stated above capture the correctness of our method in terms of trace equivalence ($\simeq$) between networks and corresponding concurrent programs.

Future Work. This paper introduces a novel formal connection between the problem of updating the configuration of a running network and minimal fence insertion in weak memory models. In the future, we plan to incorporate these results into a practical tool for synthesizing network updates with minimal waits. We also plan to develop deeper connections between programming abstractions for concurrent architectures and software-defined networks.

Acknowledgments. The authors wish to thank Fred Schneider and Keith Marzullo for suggesting we explore the connection between concurrent programming and software-defined networks. Our work is supported by the National Science Foundation under grants CNS-1111698, CNS-1413972, and SHF-1422046, SHF-1421752 and a gift from Fujitsu Labs.

References

[1] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the Verification Problem for Weak Memory Models. In *POPL*, pages 7–18, 2010.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, Aug. 2013.

[3] A. Linden and P. Wolper. A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems. In *TACAS*, pages 339–353, 2013.

[4] J. McClurg, H. Hojjat, N. Foster, and P. Černý. Efficient Synthesis of Network Updates. *CoRR*, 2014.

[5] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, pages 323–334. ACM, 2012.