Rollback-Recovery for Middleboxes

Justine Sherry∗ Peter Xiang Gao∗ Soumya Basu∗ Aurojit Panda∗
Arvind Krishnamurthy• Christian Maciocco† Maziar Manesh† João Martins◁
Sylvia Ratnasamy∗ Luigi Rizzo‡ Scott Shenker○∗

∗ UC Berkeley • University of Washington † Intel Research ◁ NEC Labs ‡ University of Pisa ○ ICSI

ABSTRACT

Network middleboxes must offer high availability, with automatic failover when a device fails. Achieving high availability is challenging because failover must correctly restore lost state (e.g., activity logs, port mappings) but must do so quickly (e.g., in less than typical transport timeout values to minimize disruption to applications) and with little overhead to failure-free operation (e.g., additional per-packet latencies of 10-100s of μs). No existing middlebox design provides failover that is correct, fast to recover, and imposes little increased latency on failure-free operations.

We present a new design for fault-tolerance in middleboxes that achieves these three goals. Our system, FTMB (for Fault-Tolerant MiddleBox), adopts the classical approach of "rollback recovery" in which a system uses information logged during normal operation to correctly reconstruct state after a failure. However, traditional rollback recovery cannot maintain high throughput given the frequent output rate of middleboxes. Hence, we design a novel solution to record middlebox state which relies on two mechanisms: (1) 'ordered logging', which provides lightweight logging of the information needed after recovery, and (2) a 'parallel release' algorithm which, when coupled with ordered logging, ensures that recovery is always correct. We implement ordered logging and parallel release in Click and show that for our test applications our design adds only 30μs of latency to median per packet latencies. Our system introduces moderate throughput overheads (5-30%) and can reconstruct lost state in 40-275ms for practical systems.

CCS Concepts

• **Networks** → **Middleboxes / network appliances;** • **Computer systems organization** → **Availability;**

Keywords

middlebox reliability; parallel fault-tolerance

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom

 ISBN 978-1-4503-3542-3/15/08…$15.00

DOI: http://dx.doi.org/10.1145/2785956.2787501

1. INTRODUCTION

Middleboxes play a crucial role in the modern Internet infrastructure – they offer an easy way to deploy new dataplane functions and are often as numerous as routers and switches [35, 59, 62]. Yet, because middleboxes typically involve proprietary monolithic software running on dedicated hardware, they can be expensive to deploy and manage.

To rectify this situation, network operators are moving towards Network Function Virtualization (NFV), in which middlebox functionality is moved out of dedicated physical boxes into virtual appliances that can be run on commodity processors [32]. While the NFV vision solves the dedicated hardware problem, it presents some technical challenges of its own. Two of the most commonly cited challenges have been performance [38, 45, 52, 55, 58] and management [33, 35, 49] with multiple efforts in both industry and academia now exploring these questions. We argue that an equally important challenge – one that has received far less attention – is that of *fault-tolerance*.

Today, the common approach to fault tolerance in middleboxes is a combination of careful engineering to avoid faults, and deploying a backup appliance to rapidly restart when faults occur. Unfortunately, neither of these approaches – alone or in combination – are ideal, and the migration to NFV will only exacerbate their problematic aspects.

With traditional middleboxes, each "box" is developed by a single vendor and dedicated to a single application. This allows vendors greater control in limiting the introduction of faults by, for example, running on hardware designed and tested for reliability (ECC, proper cooling, redundant power supply, *etc.*). This approach will not apply to NFV, where developers have little control over the environment in which their applications run and vendor diversity in hardware and applications will explode the test space. And while one might contemplate (re)introducing constraints on NFV platforms, doing so would be counter to NFV's goal of greater openness and agility in middlebox infrastructure.

The second part to how operators handle middlebox failure is also imperfect. With current middleboxes, operators often maintain a dedicated per-appliance backup. This is inefficient and offers only a weak form of recovery for the many middlebox applications that are stateful – *e.g.*, Network Address Translators (NATs), WAN Optimizers, and Intrusion Prevention Systems all maintain dynamic state about

flows, users, and network conditions. With no mechanism to recover state, the backup may be unable to correctly process packets after failure, leading to service disruption. (We discuss this further in §3.2 and quantify disruption in §6.)

Our goal in this paper is to design middleboxes that guarantee correct recovery from failures. This solution must be low-latency (*e.g.*, the additional per-packet latency under failure-free conditions must be well under 1ms) and recovery must be fast (*e.g.*, in less than typical transport timeout values). To the best of our knowledge, no existing middlebox design satisfies these goals. In addition, we would prefer a solution that is general (*i.e.*, can be applied across applications rather than having to be designed on a case-by-case basis for each individual middlebox) and passive (*i.e.*, does not require one dedicated backup per middlebox).

Our solution – FTMB– introduces new algorithms and techniques that tailor the classic approach of rollback recovery to the middlebox domain and achieves correct recovery in a general and passive manner. Our prototype implementation introduces low additional latency on failure-free operation (adding only 30μs to median per-packet latencies, an improvement of 2-3 orders of magnitude over existing fault tolerance mechanisms) and achieves rapid recovery (reconstructing lost state in between 40-275ms for practical system configurations).

The remainder of this paper is organized as follows: we discuss our assumptions and the challenges in building a fault-tolerant middlebox in §2, followed by our goals and an examination of the design space in §3. We present the design, implementation and evaluation of FTMB in §4, §5, and §6 respectively. We discuss related work in §7 and conclude with future directions in §8.

2. PROBLEM SPACE

We present our system and failure model (§2.1 and §2.2) and the challenges in building fault-tolerant middleboxes (§2.3).

2.1 System Model

Parallel implementations: We assume middlebox applications are multi-threaded and run on a multicore CPU (Figure 1). The middlebox runs with a fixed number of threads. We assume 'multi-queue' NICs that offer multiple transmit and receive queues that are partitioned across threads. Each thread reads from its own receive queue(s) and writes to its own transmit queue(s). The NIC partitions packets across threads by hashing a packet's flow identifier (*i.e.*, 5-tuple including source and destination port and address) to a queue; hence all packets from a flow are processed by the same thread and a packet is processed entirely by one thread. The above are standard approaches to parallelizing traffic processing in multicore systems [27, 37, 47, 58].

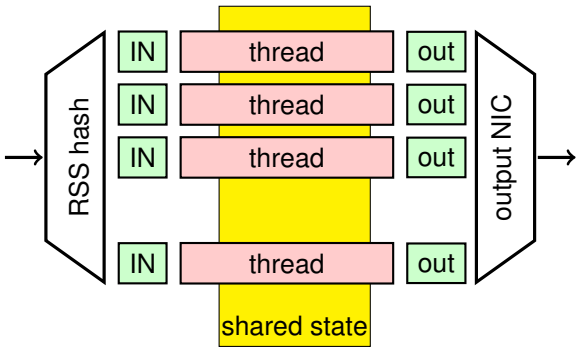


Figure 1: Our model of a middlebox application

Shared state: By shared state we mean state that is accessed across threads. In our parallelization approach, all packets from a flow are processed by a single thread so per-flow state is local to a single thread and is not shared state. However, other state may be relevant to multiple flows, and accesses to such state may incur cross-thread synchronization overheads. Common forms of shared state include aggregate counters, IDS state machines, rate limiters, packet caches for WAN optimizers, *etc*.

Virtualization: Finally, we assume the middlebox code is running in a virtualized mode. The virtualization need not be a VM per se; we could use containers [5], lightweight VMs [44], or some other form of compartmentalization that provides isolation and supports low-overhead snapshots of its content.

2.2 Failure Model

We focus on recovery from "fail-stop" (rather than Byzantine) errors, where under failure 'the component changes to a state that permits other components to detect that a failure has occurred and then stops' [57]. This is the standard failure model assumed by virtual machine fault tolerance approaches like Remus [23], Colo [28], and vSphere [11].

Our current implementation targets failures at the virtualization layer and below, down to the hardware.[1] Our solutions – and many of the systems we compare against – thus cope with failures in the system hardware, drivers, or host operating system. According to a recent study (see Figure 13 in [48]), hardware failures are quite common (80% of firewall failures, 66% of IDS failures, 74% of Load Balancer failures, and 16% of VPN failures required some form of hardware replacement), so this failure model is quite relevant to operational systems.

2.3 Challenges

Middlebox applications exhibit three characteristics that, in combination, make fault-tolerance a challenge: statefulness, very frequent non-determinism, and low packet-processing latencies.

As mentioned earlier, many middlebox applications are *stateful* and the loss of this state can degrade performance and disrupt service. Thus, we want a failover mechanism that correctly restores state such that future packets are processed as if this state were never lost (we define correctness rigorously in §3.1). One might think that this could be achieved via 'active:active' operation, in which a 'master' and a 'replica' execute on all inputs but only the mas-

[1] In §8, we discuss how emerging 'container' technologies would allow us to extend our failure model to recover from failures in the guest OS. With such extensions in place, the only errors that we would be unable to recover from are those within the middlebox application software itself.

ter's output is released to users. However, this approach fails when system execution is *non-deterministic*, because the master and replica might diverge in their internal state and produce an incorrect recovery.[2]

Non-determinism is a common problem in parallel programs when threads 'race' to access shared state: the order in which these accesses occur depends on hard-to-control effects (such as the scheduling order of threads, their rate of progress, *etc.*) and are thus hard to predict. Unfortunately, as mentioned earlier, shared state is common in middlebox applications, and shared state such as counters, caches or address pools may be accessed on a per-packet or per-flow basis leading to frequent nondeterminism.[3] In addition, nondeterminism can also arise because of access to hardware devices, including clocks and random number generators, whose return values cannot be predicted. FTMB must cope with all of these sources of nondeterminism.

As we elaborate on shortly, the common approach to accommodating non-determinism is to intercept and/or record the outcome of all potentially non-deterministic operations. However, such interception slows down normal operation and is thus at odds with the other two characteristics of middlebox applications, namely *very* frequent accesses to shared state and low packet processing latencies. Specifically, a piece of shared state may be accessed 100k-1M times per second (the rate of packet arrivals), and the latency through the middlebox should be in 10-100s of microseconds. Hence mechanisms for fault-tolerance must support high access rates and introduce extra latencies of a similar magnitude.

3. GOALS AND DESIGN RATIONALE

Building on the previous discussion, we now describe our goals for FTMB (§3.1), some context (§3.2), and the rationale for the design approach we adopt (§3.3)

3.1 Goals

A fault-tolerant middlebox design must meet the three requirements that follow.

(1) Correctness. The classic definition of correct recovery comes from Strom and Yemeni [60]: "A system recovers correctly if its internal state after a failure is consistent with the observable behavior of the system before the failure." It is important to note that reconstructed state need not be identical to that before failure. Instead, it is sufficient that the reconstructed state be one that *could* have generated the interactions that the system has already had with the external world. This definition leads to a necessary condition for correctness called "**output commit**", which is stated as follows: no output can be released to the external world until all the information necessary to recreate internal state consistent with that output has been committed to stable storage.

As we discuss shortly, the nature of this necessary information varies widely across different designs for fault-tolerance as does the manner in which the output commit property is enforced. In the context of middleboxes, the output in question is a packet and hence to meet the output commit property we must ensure that, before the middlebox transmits a packet p, it has successfully logged to stable storage all the information needed to recreate internal state consistent with an execution that would have generated p.

(2) Low overhead on failure-free operation. We aim for mechanisms that introduce no more than 10-100s of microseconds of added delay to packet latencies.

(3) Fast Recovery. Finally, recovery from failures must be fast to prevent degradation in the end-to-end protocols and applications. We aim for recovery times that avoid endpoint protocols like TCP entering timeout or reset modes.

In addition, we seek solutions that obey the following two supplemental requirements:

(4) Generality. We prefer an approach that does not require complete rewriting of middlebox applications nor needs to be tailored to each middlebox application. Instead, we propose a single recovery mechanism and assume access to the source code. Our solution requires some annotations and and automated modifications to this code. Thus, we differ from some recent work [50, 51] in not introducing an entirely new programming model, but we cannot use completely untouched legacy code. Given that middlebox vendors are moving their code from their current hardware to NFV-friendly implementations, small code modifications of the sort we require may be a reasonable middle ground.

(5) Passive Operation. We do not want to require dedicated replicas for each middlebox application, so instead we seek solutions that only need a passive replica that can be shared across active master instances.

3.2 Existing Middleboxes

To our knowledge, no middlebox design in research or deployment simultaneously meets the above goals.[4]

In research, Pico [50] was the first to address fault-tolerance for middleboxes. Pico guarantees correct recovery but does so at the cost of introducing non-trivial latency under failure-free operation – adding on the order of 8-9ms of delay per packet. We describe Pico and compare against it experimentally in §6.

There is little public information about what commercial middleboxes do and therefore we engaged in discussions with two different middlebox vendors. From our discussions, it seems that vendors do rely heavily on simply engineering the boxes to not fail (which is also the only approach one can take without asking customers to purchase a separate backup box). For example, one vendor uses only a single line of network interface cards and dedicates an entire engineering team to testing new NIC driver releases.

Both vendors confirmed that shared state commonly occurs in their systems. One vendor estimated that with their IDS implementation, a packet touches 10s of shared variables per packet, and that even their simplest devices incur at least one shared variable access per packet.

[2] Similarly, such non-determinism prevents replicated state machine techniques from providing recovery in this context.

[3] We evaluate the effects of such non-determinism in §6.

[4] Traditional approaches to reliability for routers and switches do little to address statefulness as there is no need to do so, and thus we do not discuss such solutions here.

Somewhat to our surprise, both vendors strongly rejected the idea of simply resetting all active connections after failure, citing concerns over the potential for user-visible disruption to applications (we evaluate cases of such disruption in §6). Both vendors do attempt stateful recovery but their mechanisms for this are ad-hoc and complex, and offer no correctness guarantee. For example, one vendor partially addresses statefulness by checkpointing select data structures to stable storage; since checkpoints may be both stale and incomplete (*i.e.*, not all state is checkpointed) they cannot guarantee correct recovery. After recovery, if an incoming packet is found to have no associated flow state, the packet is dropped and the corresponding connection reset; they reported using a variety of application-specific optimizations to lower the likelihood of such resets. Another vendor offers an 'active:active' deployment option but they do not address non-determinism and offer no correctness guarantees; to avoid resetting connections their IDS system 'fails open' – *i.e.*, flows that were active when the IDS failed bypass some security inspections after failure.

Both vendors expressed great interest in general mechanisms that guarantee correctness, saying this would both improve the quality of their products and reduce the time their developers spend reasoning through the possible outcomes of new packets interacting with incorrectly restored state.

However, both vendors were emphatic that correctness could not come at the cost of added latency under failure-free operation and independently cited 1ms as an upper bound on the latency overhead under failure-free operation.[5] One vendor related an incident where a trial product that added 1-2ms of delay per-packet triggered almost 100 alarms and complaints within the hour of its deployment.

Finally, both vendors emphasized avoiding the need for 1:1 redundancy due to cost. One vendor estimated a price of $250K for one of their higher-grade appliances; the authors of [58] report that a large enterprise they surveyed deployed 166 firewalls and over 600 middleboxes in total, which would lead to multi million dollar overheads if the dedicated backup approach were applied broadly.

3.3 Design Options

Our goal is to provide stateful recovery that is correct in the face of nondeterminism, yet introduces low delay under both failure-free and post-failure operation. While less explored in networking contexts, stateful recovery has been extensively explored in the general systems literature. It is thus natural to ask what we might borrow from this literature. In this section, we discuss this prior work in broad terms, focusing on general approaches rather than specific solutions, and explain how these lead us to the approach we pursued with FTMB. We discuss specific solutions and experimentally compare against them in §6.

At the highest level approaches to stateful recovery can be classified based on whether lost state is reconstructed by *replaying* execution on past inputs. As the name suggests, solutions based on 'replay' maintain a log of inputs to the system and, in the event of a failure, they recreate lost state by replaying the inputs from the log; in contrast, 'no-replay' solutions do not log inputs and never replay past execution.

As we will discuss in this section, we reject no-replay solutions because they introduce high latencies on per-packet forwarding – on the order of many milliseconds. However, replay-based approaches have their own challenges in sustaining high throughput given the output frequency of middleboxes. FTMB follows the blueprint of rollback-recovery, but introduces new algorithms for logging and output commit that can sustain high throughput.

3.4 No-Replay Designs

No-replay approaches are based on the use of system checkpoints: processes take periodic "snapshots" of the necessary system state and, upon a failure, a replica loads the most recent snapshot. However, just restoring state to the last snapshot does not provide correct recovery since all execution beyond the last snapshot is lost – *i.e.*, the output commit property would be violated for all output generated after the last snapshot. Hence, to enforce the output commit property, such systems buffer all output for the duration between two consecutive snapshots [23]. In our context, this means packets leaving the middlebox are buffered and not released to the external world until a checkpoint of the system up to the creation of the last buffered packet has been logged to stable storage.

Checkpoint-based solutions are simple but delay outputs even under failure-free operation; the extent of this delay depends on the overhead of (and hence frequency between) snapshots. Several efforts aim to improve the efficiency of snapshots – *e.g.*, by reducing their memory footprint [50], or avoiding snapshots unless necessary for correctness [28]. Despite these optimizations, the latency overhead that these systems add – in the order of many milliseconds – remains problematically high for networking contexts. We thus reject no-replay solutions.

3.5 Replay-Based Designs

In replay-based designs, the inputs to the system are logged along with any additional information (called 'determinants') needed for correct replay in the face of nondeterminism. On failure, the system simply replays execution from the log. To reduce replay time and storage requirements these solutions also use periodic snapshots as an optimization: on failure, replay begins from the last snapshot rather than from the beginning of time. Log-based replay systems can release output without waiting for the next checkpoint so long as all the inputs and events on which that output depends have been successfully logged to stable storage. This reduces the latency sensitive impact on failure-free operation making replay-based solutions better suited for FTMB.

Replay-based approaches to system *recovery* should not be confused with replay-based approaches to *debugging*. The latter has been widely explored in recent work for debugging multicore systems [16, 41, 61]. However, debug-

[5] This is also consistent with carrier requirements from the Broadband Forum which cite 1ms as the upper bound on forwarding delay (through BGN appliances) for VoIP and other latency-sensitive traffic [14].

ging systems do not provide mechanisms for output commit, the central property needed for correct recovery – they do not need to, since their aim is not to resume operation after failure. Consequently, these systems cannot be used to implement high availability. [6]

Instead, the most relevant work to our goals comes from the classic distributed systems literature from the 80s and 90s, targeting rollback-recovery for multi-process distributed systems (see [31] for an excellent survey). Unfortunately, because of our new context (a single multi-threaded server, rather than independent processes over a shared network) and performance constraints (output is released every few microseconds or nanoseconds rather than seconds or milliseconds), existing algorithms from this literature for logging and output commit cannot sustain high throughput.

With all recovery approaches, the system must check that all determinants – often recorded in the form of vector clocks [42] or dependency trees [30] – needed for a given message to be replayed have been logged before the message may be released. This check enforces the output commit property. In systems which follow an *optimistic logging* approach, this output commit 'check' requires *coordination* between all active process/threads every time output is released. This coordination limits parallelism when output needs to be released frequently. For example, in §5 we discuss a design we implemented following the optimistic approach which could sustain a maximum throughput of only 600Mbps (where many middleboxes process traffic on the order of Gbps) due to frequent cross-core coordination. Other systems, which follow a *causal logging* approach, achieve coordination-free output commit and better parallelism, but do so by permitting heavy redundancy in what they log: following the approach of one such causal system [30], we estimated that the amount of logged determinants would reach between 500Gbps-300Tbps just for a 10Gbps of packets processed on the dataplane. Under such loads, the system would have to devote far more resources to recording the logs themselves than processing traffic on the dataplane, once again limiting throughput.

Hence, instead of following a standard approach, we instead designed a new logging and output commit approach called *ordered logging with parallel release*. In the following section, we describe how our system works and why ordered logging with parallel release overcome the issues presented by previous approaches.

4. DESIGN

FTMB is a new solution for rollback recovery, tailored to the middlebox problem domain through two new solutions:

1. 'ordered logging': an efficient mechanism for representing and logging the information required for correct replay; ordered logging represents information in such a way that it is easy to verify the output commit property.

2. 'parallel release': an output commit algorithm that is simple and efficient to implement on multicore machines.

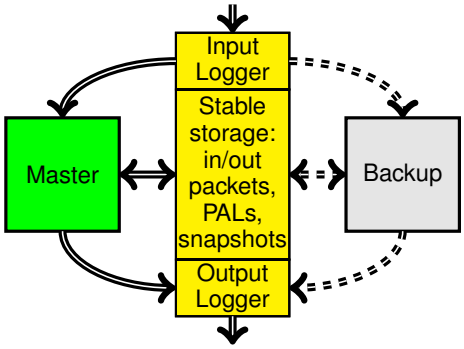


Figure 2: Architecture for FTMB.

The architecture of FTMB is shown in Figure 2. A *master* VM runs the middlebox application(s), with two *loggers* that record its input and output traffic. Periodic system snapshots are sent from the master to stable storage, and used to start a *backup* in case the master crashes. In our prototype, the master and backup are two identical servers; the input and output loggers are software switches upstream and downstream from the master node; and the stable storage is volatile memory at the downstream switch – the storage is 'stable' in that it will survive a failure at the master, even though it would not survive failure of the switch it resides on. [7]

As explained in earlier sections, the crux of ensuring correct recovery is enforcing the output commit property which, for our context, can be stated as: *do not release a packet until all information needed to replay the packet's transmission has been logged to stable storage.* Enforcing this property entails answering the following questions:

- What information must we log to resolve potential nondeterminism during replay? In the language of rollback recovery protocols this defines what the literature calls *determinants*.

- How do we log this information efficiently? This specifies how we *log* determinants.

- What subset of the information that we log is a given packet dependent on for replay? This defines an output's *dependencies*.

- How do we efficiently check when an individual packet's dependencies have been logged to stable storage? This specifies how we check whether the *output commit* requirements for an output have been met.

[6] A second question is whether or not we can adopt logging and instrumentation techniques from these systems to detect determinants. However, as we discuss experimentally in §6, most debugging approaches rely on heavyweight instrumentation (*e.g.*, using memory protection to intercept access to shared data) and often logging data that is unnecessary for our use cases (*e.g.*, all calls to malloc) – this leads to unnecessarily high overheads.

[7] There is some flexibility on the physical placement of the functions; our system can withstand the failure of either the middlebox (Master/Backup) or the node holding the saved state but not both simultaneously. We envisage the use of "bypass" NICs that fail open on detecting failure, to survive failures at the loggers [4].

We now address each question in turn and present the architecture and implementation of the resultant system in §5.

4.1 Defining Determinants

Determinants are the information we must record in order to correctly replay operations that are vulnerable to nondeterminism. As discussed previously, nondeterminism in our system stems from two root causes: races between threads accessing shared variables, and access to hardware whose return values cannot be predicted, such as clocks and random number generators. We discuss each of them below.

Shared State Variables. Shared variables introduce the possibility of nondeterministic execution because we cannot control the order in which threads access them.[8] We thus simply record the order in which shared variables are accessed, and by whom.

Each shared variable v_j is associated with its own lock and counter. The lock protects accesses to the variable, and the counter indicates the order of access. When a thread processing packet p_i accesses a shared variable v_j, it creates a tuple called Packet Access Log (PAL) that contains $(p_i, n_{ij}, v_j, s_{ij})$ where n_{ij} is the number of shared variables accessed so far when processing p_i, and s_{ij} is the number of accesses received so far by v_j.

As an example, figure 3 shows the PALs generated by the four threads (horizontal lines) processing packets A, B, C, D. For packet B, the thread first accesses variable X (which has previously been accessed by the thread processing packet A), and then variable Y (which has previously been accessed by the thread processing packet C).

Note that PALs are created independently by each thread, while holding the variable's lock, and using information (the counters) that is either private to the thread or protected by the lock itself.

Shared pseudorandom number generators are treated in the same way as shared variables, since their behavior is deterministic based on the function's seed (which is initialized in the same way during a replay) and the access order recorded in the PALs.

Clocks and other hardware. Special treatment is needed for hardware resources whose return values cannot be predicted, such as `gettimeofday()` and `/dev/random`. For these, we use the same PAL approach, but replacing the variable name and access order with the hardware accessed and the value returned. Producing these PALs does not require any additional locking because they only use information local to the thread. Upon replay, the PALs allow us to return the exact value as during the original access.

4.2 How to Log Determinants

The key requirement for logging is that PALs need to be on stable storage (on the Output Logger) before we release the packets that depend on them. While there are many options for doing so, we pursue a design that allows for fine-grained and correct handling of dependencies.

[8] Recent research [22, 25] has explored ways to reduce the performance impact of enforcing deterministic execution but their overheads remain impractically high for applications with frequent nondeterminism.

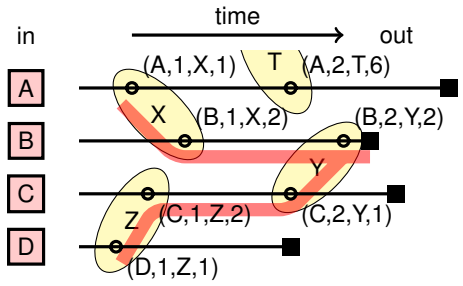


Figure 3: Four threads (black lines) process packets A, B, C, D. As time goes (left to right), they access (circles) shared variables X, Y, Z, T generating the PALs in parentheses. The red tree indicates the dependencies for packet B.

We make two important design decisions for how logging is implemented. The first is that PALs are *decoupled* from their associated data packet and communicated separately to the output logger. This is essential to avoid introducing unnecessary dependencies between packets. As an example, packet B in the figure depends on PAL $(A, 1, X, 1)$, but it need not be delayed until the completion of packet A, (which occurs much later) – it should only be delayed until $(A, 1, X, 1)$ has been logged.

The second decision has to do with *when* PALs are placed in their outgoing PAL queue. **We require that PALs be placed in the output queue before releasing the lock associated to the shared variable they refer to.** This gives two guarantees: i) when p_i is queued, all of its PALs are already queued; and ii) when a PAL for v_j is queued, all previous PALs for the same variable are already in the output queues for this or other threads. We explain the significance of these properties when we present the output commit algorithm in §4.4.

4.3 Defining a Packet's Dependencies

During the replay, the replica must evolve in the same way as the master. For a shared variable v_j accessed while processing p_i, this can happen only if i) the variable has gone through the same sequence of accesses, and ii) the thread has the same internal state. These conditions can be expressed recursively in terms of the PALs: each PAL (p_i, n, v_j, m) in turn has up to two dependencies: one **per-packet** $(p_i, n-1, v_k, s_{ik})$, *i.e.*, on its predecessor PAL for p_i, and one **per-variable** $(p_{i'}, n', v_j, m-1)$, *i.e.*, on its predecessor PAL for v_j, generated by packet $p_{i'}$. A packet depends on its last PAL, and from that we can generate the tree of dependencies; as an example, the red path in the figure represents the dependencies for packet B.

We should note that the recursive dependency is essential for correctness. If, for instance, packet B in the figure were released without waiting for the PAL $(D, 1, Z, 1)$, and the thread generating that PAL crashed, during the replay we could not adequately reconstruct the state of the shared variables used while processing packet B.

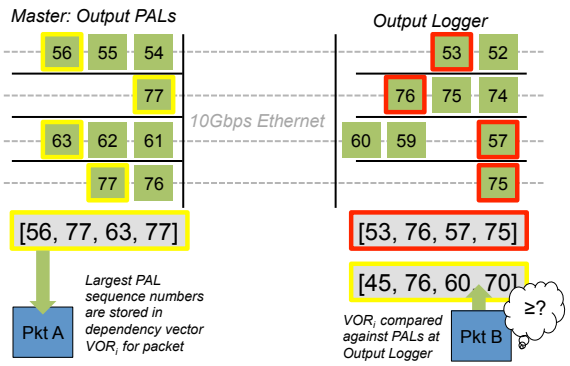


Figure 4: Parallel release. Each PAL is assigned a sequence number identifying *when it was generated* **within that thread; a packet is released from the output logger if all PALs that were queued before it (on any thread) have been logged.**

4.4 Output Commit

We now develop an algorithm that ensures we do not release p_i until all PALs corresponding to p_i's dependencies have arrived at the output logger. This output commit decision is implemented at the output logger. The challenge in this arises from the parallel nature of our system. Like the master, our output logger is multi-threaded and each thread has an independent queue. As a result, the PALs corresponding to p_i's dependencies may be distributed across multiple per-thread queues. We must thus be careful to minimize cache misses and avoid the use of additional synchronization operations.

Rejected Design: Fine-grained Tracking

The straightforward approach would be to explicitly track individual packet and PAL arrivals at the output logger and then release a packet p_i after all of its PAL dependencies have been logged. Our first attempt implemented a 'scoreboard' algorithm that did exactly this at the output logger. We used two matrices to record PAL arrivals: (i) SEQ$[i, j]$ which stores the sequence number of p_i at v_j and (ii) PKT$[j, k]$, the identifier of the packet that accessed v_j at sequence number s_k. These data structures contain all the information needed to check whether a packet can be released. We designed a lock-free multi-threaded algorithm that provably released data packets immediately as their dependencies arrived at the middlebox; however, the overhead of cache contention in reading and updating the scoreboard resulted in poor throughput. Given the two matrices described above, we can expect O(nc) cache misses per packet release, where n is the number of shared variables and c the number of cores (we omit details due to space considerations). Despite optimizations, we find that explicitly tracking dependencies in the above fashion will result in the scoreboard becoming the bottleneck for simple applications.

Parallel release of PALs

We now present a solution that is slightly more coarse-grained, but is amenable to a parallel implementation with very limited overhead. Our key observation here is that the rules chosen to queue PALs and packets *guarantee* that both the **per-packet** and **per-variable** dependencies for a given packet are already queued for release on some thread before the packet arrives at the output queue on its own thread. This follows from the fact that the PAL for a given lock access is always queued *before* the lock is released. Hence, we only need to transfer PALs and packets to the output logger in a way that preserves the ordering between PALs and data packets.

This is achieved with a simple algorithm run between the Master and the Output Logger, illustrated in Fig. 4. Each thread on the Master maps 'one to one' to an ingress queue on the Output Logger. PALs in each queue are transferred as a sequential stream (similar to TCP), with each PAL associated to an per-queue sequence number. This replaces the second entry in the PAL, which then does not need to be stored. Each thread at the Master keeps track of MAX, the maximum sequence number that has been assigned to any PAL it has generated.

On the Master: Before sending a data packet from its queue to the output logger, each thread on the master *reads* the current MAX value at all other threads and creates a vector clock VOR which is associated with the packet. It then reliably transfers the pending PALs in its queue, followed by the data packets and associated vector clocks.

On the Output Logger: Each thread continuously receives PALs and data packets, requesting retransmissions in the case of dropped PALs. When it receives a PAL, a thread updates the value MAX representing the highest sequence number such that it has received all PALs prior to MAX. On receiving a data packet, each thread *reads* the value MAX over all other threads, comparing each with the vector clock VOR. Once all values $\text{MAX}_i \geq \text{VOR}_i$, the packet can be released.

Performance

Our parallel release algorithm is efficient because i) threads on the master and the output logger can run in parallel; ii) there are no write-write conflicts on the access to other queues, so memory performance does not suffer much; iii) the check to release a packet requires a very small constant time operation; iv) when batching is enabled, all packets released by the master in the same batch can use the same vector clock, resulting in very small overhead on the link between the master and the output logger and amortizing the cost of the 'check' operation.

5. SYSTEM IMPLEMENTATION

We present key aspects of our implementation of FTMB. For each, we highlight the performance implications of adding FTMB to a regular middlebox through qualitative discussion and approximate back-of-the-envelope estimates; we present experimental results with our prototype in §6.

The logical components of the architecture are shown in Figure 2. Packets flow from the Input Logger (IL), to the Master (M), to the Output Logger (OL). FTMB also needs a Stable Storage (SS) subsystem with enough capacity to store

the state of the entire VM, plus the packets and PALs accumulated in the IL and OL between two snapshots. In our implementation the IL, OL and SS are on the same physical machine, which is expected to survive when M crashes.

To estimate the amount of storage needed we can assume a snapshot interval in the 50–200 ms range (§6), and input and output traffic limited by the link's speed (10–40 Gbit/s). We expect to cope with a large, but not overwhelming PAL generation rate; *e.g.*, in the order of 5 M PALs/s (assuming an input rate of 1.25M packets/second and 5 shared state accesses per packet).

5.1 Input Logger

The main role of the IL is to record input traffic since the previous snapshot, so that packets can be presented in the same order to the replica in case of a replay.

The input NIC on the IL can use standard mechanisms (such as 5-tuple hashing on multiqueue NICs) to split traffic onto multiple queues, and threads can run the IL tasks independently on each queue. Specifically, on each input queue, the IL receives incoming packets, assigns them sequence numbers, saves them into stable storage, and then passes them reliably to the Master.

Performance implications: The IL is not especially CPU intensive, and the bandwidth to communicate with the master or the storage is practically equal to the input bandwidth: the small overhead for reliably transferring packets to the Master is easily offset by aggregating small frames into MTU-sized segments.

It follows that the only effect of the IL on performance is the additional (one way) latency for the extra hop the traffic takes, which we can expect to be in the 5–10μs range [34].

5.2 Master

The master runs a version of the Middlebox code with the following modifications:

- the input must read packets from the reliable stream coming from the IL instead of individual packets coming from a NIC;
- the output must transfer packets to the output queue instead of a NIC.
- access to shared variables is protected by locks, and includes calls to generate and queue PALs;
- access to special hardware functions (timers, etc.) also generates PALs as above.

A shim layer takes care of the first two modifications; for a middlebox written using Click, this is as simple as replacing the `FromDevice` and `ToDevice` elements. We require that developers annotate shared variables at the point of their declaration. Given these annotations, we automate the insertion of the code required to generate PALs using a custom tool inspired by generic systems for data race detection [56].

Our tool uses LLVM's [43] analysis framework (also used in several static analysis tools including the Clang Static Analyzer [3] and KLEE [19]) to generate the call graph for the middlebox. We use this call graph to record the set of locks held while accessing each shared variable in the middlebox. If all accesses to the shared variable are protected by a common lock, we know that there are no contended accesses to the variable and we just insert code to record and update the PAL. Otherwise we generate a "protecting" lock and insert code that acquires the lock before any accesses, in addition to the code for updating the PALs. Note that because the new locks never wrap another lock (either another new lock or a lock in the original source code), it is not possible for this instrumentation to introduce deadlocks [17, 21]. Since we rely on static analysis, our tool is conservative, *i.e.* it might insert a protecting lock even when none is required.

FTMB is often compatible with lock-free optimizations. For example, we implemented FTMB to support seqlocks [13], which are used in multi-reader/single-writer contexts. seqlocks use a counter to track what 'version' of a variable a reader accessed; this version number replaces s_{ij} in the PAL.

Performance implications: the main effect of FTMB on the performance of the Master is the cost of PAL generation, which is normally negligible unless we are forced to introduce additional locking in the middlebox.

5.3 Output Logger

The Output Logger cooperates with the Master to transfer PALs and data packets and to enforce output commit. The algorithm is described in §4.4. Each thread at M transports packets with a unique header such that NIC hashing at OL maintains the same affinity, enforcing a one-to-one mapping between an eggress queue on M to an ingress queue on OL.

The traffic between M and OL includes data packets, plus additional information for PALs and vector clocks. As a very coarse estimate, even for a busy middlebox with a total of 5 M PALs and vector clocks per second, assuming 16 bytes per PAL, 16 bytes per vector clock, the total bandwidth overhead is about 10% of the link's capacity for a 10 Gbit/s link.

Performance implications: once again the impact of FTMB on the OL is more on latency than on throughput. The minimum latency to inform the OL that PALs are in stable storage is the one-way latency for the communication. On top of this, there is an additional latency component because our output commit check requires *all queued PALs* to reach the OL before the OL releases a packet. In the worst case a packet may find a full PAL queue when computing its vector clock, and so its release may be delayed by the amount of time required to transmit a full queue of PALs. Fortunately, the PAL queue can be kept short *e.g.*, 128 slots each, without any adverse effect on the system (PALs can be sent to the OL right away; the only reason to queue them is to exploit batching). For 16-byte PALs, it takes less than 2μs of link time to drain one full queue, so the total latency introduced by the OL and the output commit check is in the 10-30μs range.

5.4 Periodic snapshots

FTMB takes periodic snapshots of the state of the Master, to be used as a starting point during replay, and avoid unbounded growth of the replay time and input and output

logs size. Checkpointing algorithms normally freeze the VM completely while taking a snapshot of its state.

Performance implications: The duration of the freeze, hence the impact on latency, has a component proportional to the number of memory pages modified between snapshots, and inversely proportional to bandwidth to the storage server. This amounts to about 5μs for each 4 Kbyte page. on a 10 Gbit/s link, and quickly dominates the fixed cost (1-2ms) for taking the snapshot. However, a worst case analysis is hard as values depend on the (wildly variable) number of pages modified between snapshots. Hence it is more meaningful to gauge the additional latency from the experimental values in §6 and the literature in general [23].

5.5 Replay

Finally, we describe our implementation of replay, when a Replica VM starts from the last available snapshot to take over a failed Master. The Replica is started in "replay mode", meaning that the input is fed (by the IL) from the saved trace, and threads use the PALs to drive nondeterministic choices.

On input, the threads on the Replica start processing packets, discarding possible duplicates at the beginning of the stream. When acquiring the lock that protects a shared variable, the thread uses the recorded PALs to check whether it can access the lock, or it has to block waiting for some other thread that came earlier in the original execution. The information in the PALs is also used to replay hardware related non deterministic calls (clocks, etc.). Of course, PALs are not generated during the replay.

On output, packets are passed to the OL, which discards them if a previous instance had been already released, or pass it out otherwise (*e.g.*, copies of packets still in the Master when it crashed, even though all of their dependencies had made it to the OL). A thread exits replay mode when it finds that there are no more PALs for a given shared variable. When this happens, it starts behaving as the master, i.e. generate PALs, compute output dependencies, *etc.*

Performance implications: other than having to re-run the Middlebox since the last snapshot, operation speed in replay mode is comparable to that in the original execution. §6.2 presents some experimental results. Of course, the duration of service unavailability after a failure also depends on the latency of the failure detector, whose discussion is beyond the scope of this paper.

6. EVALUATION

We added FTMB support into 7 middlebox applications implemented in Click: one configuration comes from industry, five are research prototypes, and one is a simple 'blind forwarding' configuration which performs no middlebox processing; we list these examples in Table 1.

Our experimental setup is as follows. FTMB uses Xen 4.2 at the master middlebox with Click running in an OpenSUSE VM, chosen for its support of fast VM snapshotting [6]. We use the standard Xen bridged networking backend; this backend is known to have low throughput and substantial recent work aims to improve throughput and latency to virtual machines, *e.g.*, through netmap+xennet [45, 52] or dpdk+virtio [38, 55]. However, neither of these latter systems yet supports seamless VM migration. We thus built two prototypes: one based on the Xen bridged networking backend which runs at lower rates (100Mbps) but is complete with support for fast VM snapshots and migration, and a second prototype that uses netmap+xennet and scales to high rates (10Gbps) but lacks snapshotting and replay. We primarily report results from our complete prototype; results for relevant experiments with the high speed prototype were qualitatively similar.

Middlebox	LOC	SVs	Elts	Source
Mazu-NAT	5728	3	46	Mazu Networks [7]
WAN Opt.	5052	2	40	Aggarwal et al. [15]
BW Monitor	4623	251	41	Custom
SimpleNAT	4964	2	42	Custom
Adaptive LB	5058	1	42	Custom
QoS Priority	5462	3	56	Custom
BlindFwding	1914	0	24	Custom

Table 1: Click configurations used in our experiments, including Lines of Code (LOC), Shared Variables (SVs), number of Elements (Elts), and the author/origin of the configuration.

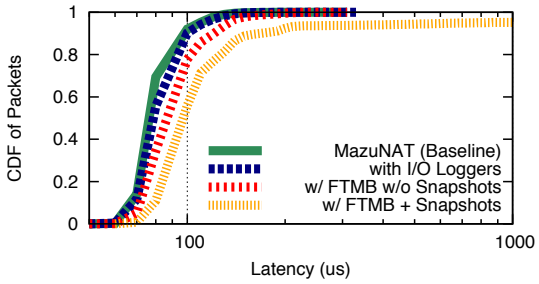


Figure 5: Local RTT with and without components of FTMB enabled.

We ran our tests on a local network of servers with 16-core Intel Xeon EB-2650 processors at 2.6Ghz, 20MB cache size, and 128GB memory divided across two NUMA nodes. For all experiments shown, we used a standard enterprise trace as our input packet stream [26]; results are representative of tests we ran on other traces.

We first evaluate the FTMB's latency and bandwidth overheads under failure-free operation (§6.1). We then evaluate recovery from failure (§6.2).

6.1 Overhead on Failure-free Operation

How does FTMB impact packet latency under failure-free operation? In Figure 5, we present the per-packet latency through a middlebox over the local network. A packet source sends traffic (over a logging switch) to a VM running a MazuNAT (a combination firewall-NAT released by Mazu Networks [7]), which loops the traffic back to the packet generator. We measure this RTT. To test FTMB, we first show the base latency with (a) just the MazuNAT, (b) the MazuNAT with I/O logging performed at the upstream/-downstream switch, (c) the MazuNAT with logging, PAL-

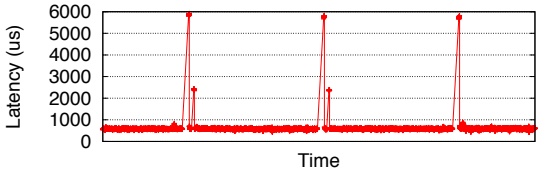


Figure 6: Testbed RTT over time.

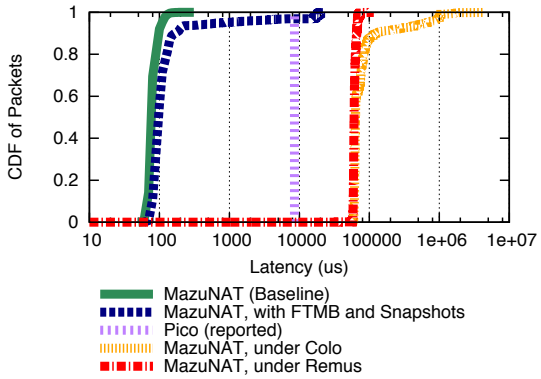


Figure 7: Local RTT with FTMB and other FT systems.

instrumented locks, parallel release for the output commit condition and (d) running the MazuNAT with all our fault tolerance mechanisms, including VM checkpointing every 200ms. Adding PAL instrumentation to the middlebox locks in the MazuNAT has a negligible impact on latency, increasing 30μs over the baseline at the median, leading to a 50th percentile latency of 100μs.[9] However, adding VM checkpointing does increase latency, especially at the tail: the 95th %-ile is 810μs, and the 99th %-ile s 18ms.

To understand the cause of this tail latency, we measured latency against time using the Blind Forwarding configuration. Figure 6 shows the results of this experiment: we see that the latency spikes are periodic with the checkpoint interval. Every time we take a VM snapshot, the virtual machine suspends temporarily, leading to a brief interval where packets are buffered as they cannot be processed. As new hardware-assisted virtualization techniques improve [1, 20] we expect this penalty to decrease with time; we discuss these opportunities further in §8.

How does the latency introduced by FTMB compare to existing fault-tolerance solutions? In Figure 7, we compare FTMB against three proposals from the research community: Pico [50], Colo [28], and Xen Remus [23]. Remus and Colo are general no-replay solutions which can provide fault tolerance for any VM-based system running a standard operating system under x86. Remus operates by checkpointing and buffering output until the next checkpoint completes; this results in a median latency increase for the MazuNAT by over 50ms. for general applications Colo can offer much lower latency overhead than Remus: Colo allows two copies of a virtual machine to run side-by-side in "lock step". If their output remains the same, the two virtual machines are considered identical; if the two outputs differ, the system

[9] In similar experiments with our netmap-based prototype we observe a median latency increase of 25μs and 40μs over the baseline at forwarding rates of 1Gbps and 5Gbps respectively, both over 4 cores.

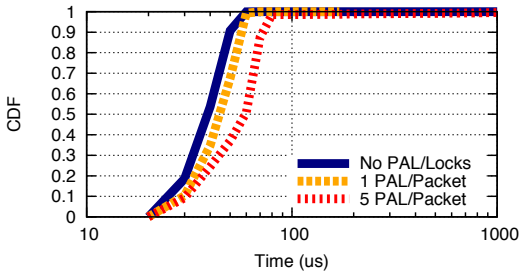


Figure 8: Testbed RTT with increasing PALs/packet.

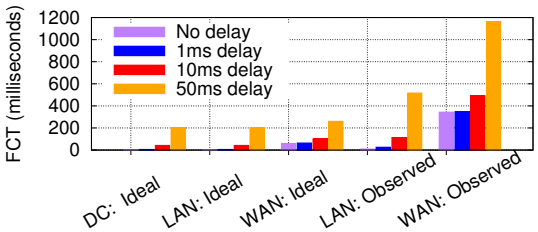


Figure 9: Ideal [46] and observed page load times when latency is artificially introduced in the network.

forces a checkpoint like Remus. Because multi-threaded middleboxes introduce substantial nondeterminism, though, Colo cannot offer us any benefits over Remus: when we ran the MazuNAT under Colo, it checkpointed just as frequently as Remus would have, leading to an equal median latency penalty.

Pico is a no-replay system similar to Remus but tailored to the middlebox domain by offering a custom library for flow state which checkpoints packet processing state only, but *not* operating system, memory state, *etc.*, allowing for much lighter-weight and therefore faster checkpoint. The authors of Pico report a latency penalty of 8-9ms in their work which is a substantial improvement over Colo and Remus, but still a noticeable penalty due to the reliance on packet buffering until checkpoint completion.

How does inserting PALs increase latency? To measure the impact of PALs over per-packet latency, we used a toy middlebox with a simple pipeline of 0, 1, or 5 locks and ran measurements with 500-byte packets at 1Gbps with four threads dedicated to processing in our DPDK testbed. Figure 8 shows the latency distributions for our experiments, relative to a baseline of the same pipeline with no locks. At 5 PALS/Locks per packet, latency increases to 60μs with 5 PALS/Locks per packet, relative to a median latency under 40μs in the baseline – an increase of on average 4μs per PAL/Lock per packet. Note that this latency figure includes both the cost of PAL creation and lock insertion; the worst case overhead for FTMB is when locks are not already present in the base implementation.

How much does latency matter to application performance? We measured the impact of inflated latency on Flow Completion Times (FCTs) with both measurements and modeling. In Figure 9, we show flow completion times for a 2MB flow (representative of web page load sizes) given the flow completion time model by Mittal et al. [46] marked

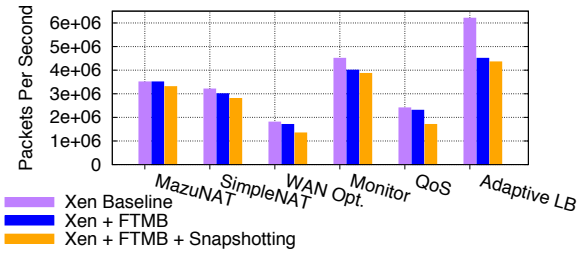


Figure 10: Impact of FTMB on forwarding plane throughput.

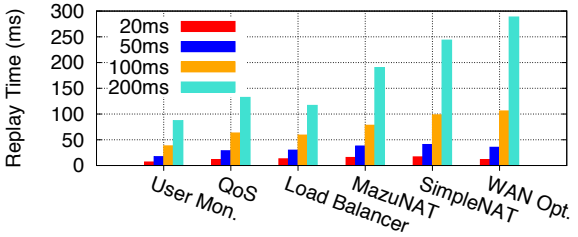


Figure 11: Time to perform replay with varying checkpoint intervals and middlebox configurations.

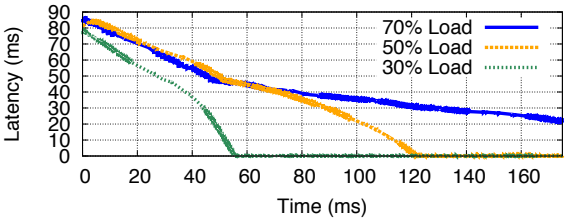


Figure 12: Packet latencies post-replay.

as 'Ideal'. Marked as 'Observed', we downloaded the Alexa top-1000 [2] web pages over a LAN and over a WAN and used `tc` to inflate the latency by the same amounts. In both the datacenter and LAN cases, adding 10ms of latency on the forward and reverse path increases flow completion times to 20× the original in the simulated case; in the experimental LAN case it increased FCT to 10×. In the WAN case, page load times increased to 1.5× by adding 10ms of latency from a median of 343ms to 492ms. An experiment by Amazon shows that every 100ms of additional page load time their customers experienced costs them 1% in sales [40].

Given these numbers in context, we can return to Figure 7 and see that solutions based on Colo, Pico, or Remus would *noticeably* harm network users' quality of experience, while FTMB, with introduced latency typically well under 1ms, would have a much weaker impact.

How much does FTMB impact throughput under failure-free operation? Figure 10 shows forwarding plane throughput in a VM, in a VM with PAL instrumentation, and running complete FTMB mode with both PAL instrumentations and periodic VM snapshotting. To emphasize the extra load caused by FTMB, we ran the experiment with locally sourced traffic and dropping the output. Even so, the impact is modest, as expected (see §5.2). For most configurations, the primary throughput penalty comes from snapshotting rather than from PAL insertion. The MazuNAT and SimpleNat saw a total throughput reduction of 5.6% and 12.5% respectively. However, for the Monitor and the Adaptive Load Balancer, PAL insertion was the primary overhead, causing a 22% and 30% drop in throughput respectively. These two experience a heavier penalty since typically they have no contention for access to shared state variables: the tens of nanoseconds required to generate a PAL for these middleboxes is a proportionally higher penalty than it is for middleboxes which spend more time per-packet accessing complex and contended state.

We ran similar experiments with Remus and Colo, where throughput peaked in the low hundreds of *Kpps*. We also ran experiments with Scribe [41], a publicly-available system for record and replay of general applications, which aims to *automatically* detect and record data races using page protection. This costs about 400us per lock access due to the overhead of page faults.[10] Using Scribe, a simple two-threaded Click configuration with a single piece of shared state stalled to a forwarding rate of only 500 packets/second.

[10] Measured using the Scribe demo image in VirtualBox.

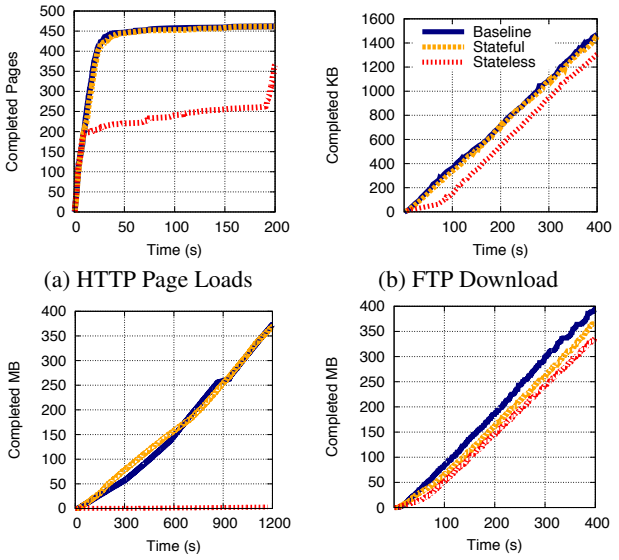
6.2 Recovery

How long does FTMB take to perform replay and how does replay impact packet latencies? Unlike no-replay systems, FTMB adds the cost of replay. We measure the amount of time required for replay in Fig. 11. We ran these experiments at 80% load (about 3.3 Mpps) with periodic checkpoints of 20, 50, 100, and 200ms.

For lower checkpoint rates, we see two effects leading to a replay time that is actually *less* than the original checkpoint interval. First, the logger begins transmitting packets to the replica as soon as replay begins – while the VM is loading. This means that most packets are read pre-loaded to local memory, rather than directly from the NIC. Second, the transmission arrives at almost 100% of the link rate, rather than 80% load as during the checkpoint interval.

However, at 200ms, we see a different trend: some middleboxes that make frequent accesses to shared variable have a *longer* replay time than the original checkpoint interval because of the overhead of replaying lock accesses. Recall that when a thread attempts to access a shared-state variable during replay, it will spin waiting for its 'turn' to access the variable and this leads to slowed execution.

During replay, new packets that arrive must be buffered, leading to a period of increased queueing delays after execution has resumed. In Figure 12, we show per-packet latencies for packets that arrive post-failure for MazuNAT at different load levels and replay times between 80-90ms. At 30%-load, packet latencies return to their normal sub-millisecond values within 60ms of resumed execution. As expected recovery takes longer at higher loads: at 70% load per-packet latency remains over 10ms even at 175ms, and the latencies do not decrease to under a millisecond until past 300ms after execution has resumed.



(a) HTTP Page Loads (b) FTP Download

(c) Torrent (Evolve) (d) Torrent(Ubuntu)

Figure 13: Application performance with and without state restoration after recovery. Key (top right) is same for all figures.

Is stateful failover valuable to applications? Perhaps the simplest approach to recovering from failure is simply to bring up a backup from 'cold start', effectively wiping out all connection state after failure: i.e., recovery is *stateless*. To see the impact of stateless recovery on real applications, we tested several applications over the wide area with a NAT which either (a) did not fail (our baseline), (b) went absent for 300ms,[11] during which time traffic was buffered (this represents stateful recovery), or (c) flushed all state on failure (representing stateless recovery). Figure 13 shows the time to download 500 pages in a 128-thread loop from the Alexa-top US sites, percentage file completion over time for a large FTP download, and percentage file completion for two separate BitTorrent downloads. In all three configurations, stateful recovery performs close to the performance of the baseline. For stateless recovery over the HTTP connections, we see a sharp knee corresponding to the connection reset time: 180 seconds[12]. The only application with little impact under stateless recovery is one of the BitTorrent downloads – however, the other BitTorrent download failed almost entirely and the client had to be restarted! The torrent which failed had only 10 available peers and, when the connections were reset, the client assumed that the peers had gone offline. The other torrent had a large pool of available peers and hence could immediately reconnect to new peers.

Our point in these experiments is not to suggest that applications are fundamentally incapable of rapid recovery in scenarios of stateless recovery, but simply that many existing applications do not.

[11] We picked 300ms as a conservative estimate of recovery time; our results are not sensitive to the precise value.

[12] Firefox, Chrome, and Opera have reset times of 300 seconds, 50 seconds, and 115 seconds respectively.

7. RELATED WORK

We briefly discuss the three lines of work relevant to FTMB, reflecting the taxonomy of related work introduced in §2.

First are no-replay schemes. In §6 we described in detail three recent systems – Remus, Pico and Colo – that adopt this approach and compared FTMB to them experimentally.

The second are solutions for rollback recovery from the distributed systems literature. The literature includes a wide range of protocol proposals (we refer the reader to Elnozahy et al. [31] for an excellent survey); however, to our knowledge, there is no available system implementation that we can put to the test for our application context.[13] More generally, as mentioned earlier, the focus on distributed systems (as opposed to a parallel program on a single machine) changes the nature of the problem in many dimensions, such as: the failure model (partial vs. complete failure), the nature of non-determinism (primarily the arrival and sending order of messages at a given process vs. threads that 'race' to access the same variable), the frequency of output (for us, outputs are generated at a very high rate) and the frequency of nondeterminism (per-packet for us), and where the performance bottlenecks lie (for us, in the logging and output commit decision). These differences led us to design new solutions that are simpler and more lightweight than those found in the literature.

The final class of solutions are the multicore record-and-replay systems used for debugging. These do not implement output commit. We discussed these solutions in broad terms in §2 and evaluated one such system (Scribe) in §6.

In the remainder of this section we briefly review a few additional systems.

Hypervisor-based Fault Tolerance [18] was an early, pioneering system in the 90s to implement fault-tolerance over arbitrary virtual machines; their approach did not address multicore systems, and required synchronization between the master and replica for every nondeterministic operation.

SMP Revirt [29] performs record-and-replay over Xen VMs; unlike FTMB SMPRevirt is hence fully general. As in Scribe, SMP ReVirt uses page protection to track memory accesses. For applications with limited contention, the authors report a 1.2-8x slowdown, but for so-called "racy" applications (like ours) with tens or hundreds of thousands of faults per second we expect results similar to those of Scribe.

Eidetic Systems [24] allow a user to replay any event in the system's history – on the scale of even years. They achieve very low overheads for their target environment: end user desktops. However, the authors explicitly note that their solutions do not scale to racy and high-output systems.

R2 [36] logs a cut in an application's call graph and introduces detailed logging of information flowing across the cut using an R2 runtime to intercept syscalls and underlying libraries; the overhead of their interception makes them poorly suited to our application with frequent nondeterminism.

ODR [16] is a general record-and-replay system that pro-

[13] In their survey paper, Elnozahy et al. state that, in practice, log-based rollback-recovery has seen little adoption due to the complexity of its algorithms.

vides output determinism: to reduce runtime overhead ODR foregoes logging all forms of nondeterminism and instead searches the space of possible executions during replay. This can result in replay times that are several orders of magnitude higher than the original execution (in fact, the search space is NP hard). This long replay time is not acceptable for applications looking to recover from a failure (as opposed to debugging post-failure).

8. DISCUSSION

We presented FTMB, a system for rollback recovery which uses ordered logging and parallel release for low overhead middlebox fault-tolerance. We showed that FTMB imposes only 30μs of latency for median packets through an industry-designed middlebox. FTMB has modest throughput overheads, and can perform replay recovery in 1-2 wide area RTTs. Before closing, we discuss the growing NFV software ecosystem and how FTMB fits into this future.

User-Level Packet Processing Stacks. Significant efforts in industry and research aim to allow for fast packet-processing in user space (*e.g.* netmap [52], DPDK [38]) in and out of virtual machines [39,45,53–55]. At present, none of these new systems support seamless virtual machine migration. This support will be required as NFV stacks become more widely deployed; once these systems do support migration they will be compatible with FTMB.

New Virtualization Techniques and Fast Checkpointing. Linux Containers [5] offer finer-grained virtualization than Xen, offering processes isolation while sharing operating system components. Containers enable cheaper snapshotting, as less memory is copied per snapshot. Several ongoing efforts are exploring hardware and software support for faster snapshotting [1, 20] which will improve migration and reliability systems [11, 23] thereby improving FTMB's tail latencies introduced by VM suspension.

Varied Operating Systems and Hardware Components. One of the oft-touted benefits of NFV [32] is the potential for a wider range of diversity in system software stacks and hardware; wider at least, than today's closed-down, all-in-one 'appliances'. With this change comes the opportunity to extend the failure model assumed by FTMB and other systems (*e.g.* [23]): by running different drivers, NICs, and operating systems outside of the virtualization layer, one may be able to protect against not only hardware and software failures, but bugs fundamental to the choice of driver, NIC, OS, *etc.*, by failing over to a machine with entirely different components.

Diverse Middlebox Software. The Open Source Community offers software implementations of many middleboxes; *e.g.*, Snort [8], Bro [10], Vyatta [12], and Squid [9], as does the broader industry. While we experimented with Click-based middlebox implementations, we expect our techniques should be equally applicable to these other systems. In addition, vendors of hardware-based appliances are increasingly re-architecting their products as software-only implementations. Hence we expect the potential application of FTMB to only grow in the future.

If the NFV ecosystem continues to gain traction with the above software trends, it will need a practical, low-overhead solution for reliability. We envision FTMB's approach – ordered logging and parallel release – to be that solution.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers of the SIGCOMM program committee and our shepherd Jeff Chase for their thoughtful feedback on this paper. We thank the middlebox vendors we spoke with for helpful discussions about FTMB, reliability practices, and state of the art network appliances. Jiawei Chen and Eddie Dong at Intel kindly shared the Colo source code and helped us deploy it in our lab at Berkeley. Kay Ousterhout provided feedback on many iterations of this paper. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1106400. This work was in part made possible by generous financial support and technical feedback from Intel Research.

10. REFERENCES

[1] A Peek into Extended Page Tables. https://communities.intel.com/community/itpeernetwork/datastack/blog/2009/06/02/a-peek-into-extended-page-tables.
[2] Alexa: Actionable Analytics for the Web. http://www.alexa.com/.
[3] Clang Static Analyzer. http://clang-analyzer.llvm.org/.
[4] Intel PRO/1000 Quad Port Bypass Server Adapters. http://www.intel.com/content/www/us/en/network-adapters/gigabit-network-adapters/pro-1000-qp.html.
[5] LXC - Linux Containers. https://linuxcontainers.org/.
[6] Remus PV domU Requirements. http://wiki.xen.org/wiki/Remus_PV_domU_requirements.
[7] Riverbed Completes Acquisition of Mazu Networks. http://www.riverbed.com/about/news-articles/press-releases/riverbed-completes-acquisition\-of-mazu-networks.html.
[8] Snort IDS. https://www.snort.org/.
[9] Squid: Optimising Web Delivery. http://www.squid-cache.org/.
[10] The Bro Network Security Monitor. https://www.bro.org/.
[11] VMWare vSphere. https://www.vmware.com/support/vsphere.
[12] Vyatta. http://www.vyatta.com.
[13] Wikipedia:seqlock. http://en.wikipedia.org/wiki/Seqlock.
[14] Multi-Service Architecture and Framework Requirements. http://www.broadband-forum.org/technical/download/TR-058.pdf, 2003.
[15] A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *Proc. USENIX NSDI*, 2010.
[16] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proc. ACM SOSP*, 2009.
[17] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.
[18] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *Proc. ACM SOSP*, 1995.
[19] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. USENIX OSDI*, 2008.
[20] J. Chung, J. Seo, W. Baek, C. CaoMinh, A. McDonald, C. Kozyrakis, and K. Olukotun. Improving Software Concurrency with Hardware-assisted Memory Snapshot. In *Proc. ACM SPAA*, 2008.
[21] E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.
[22] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A Practical Runtime for

Deterministic, Stable, and Reliable Threads. In *Proc. ACM SOSP*, 2013.

[23] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. USENIX NSDI*, 2008.

[24] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic Systems. In *Proc. USENIX OSDI*, 2014.

[25] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Proc. ACM ASPLOS*, 2009.

[26] Digital Corpora. 2009-M57-Patents packet trace. http://digitalcorpora.org/corp/nps/scenarios/2009-m57-patents/net/.

[27] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Proc. ACM SOSP*, 2009.

[28] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service. In *Proc. ACM SoCC*, 2013.

[29] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proc. ACM SIGPLAN/SIGOPS VEE*, 2008.

[30] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992.

[31] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[32] European Telecommunications Standards Institute. NFV Whitepaper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[33] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *Proc. USENIX NSDI*, 2014.

[34] M. Flajslik and M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *Proc. USENIX ATC*, 2013.

[35] A. Gember, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proc. ACM SIGCOMM*, 2014.

[36] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proc. USENIX OSDI*, 2008.

[37] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated Software Router. In *Proc. ACM SIGCOMM*, 2010.

[38] Intel. Data Plane Development Kit. http://dpdk.org/.

[39] Intel. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html.

[40] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, 2007.

[41] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proc. ACM SIGMETRICS*, 2010.

[42] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[43] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. IEEE CGO*, 2004.

[44] J. R. Lorch, A. Baumann, L. Glendenning, D. Meyer, and A. Warfield. Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services. In *Proc. USENIX NSDI*.

[45] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. USENIX NSDI*, 2014.

[46] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively Cautious Congestion Control. In *Proc. USENIX NSDI*, 2014.

[47] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proc. ACM EuroSys*, 2012.

[48] R. Potharaju and N. Jain. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. In *Proc. ACM IMC*, 2013.

[49] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. ACM SIGCOMM*, 2013.

[50] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. ACM SoCC*, 2013.

[51] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. USENIX NSDI*, 2013.

[52] L. Rizzo. netmap: a Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC*, 2012.

[53] L. Rizzo and G. Lettieri. Vale: a Switched Ethernet for Virtual Machines. In *Proc. ACM CoNEXT*, 2012.

[54] L. Rizzo, G. Lettieri, and V. Maffione. Speeding Up Packet I/O in Virtual Machines. In *ACM/IEEE ANCS*, pages 47–58, 2013.

[55] R. Russell. virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM OSR*, 42(5):95–103, 2008.

[56] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. ACM SOSP*, 1997.

[57] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[58] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. USENIX NSDI*, 2012.

[59] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. ACM SIGCOMM*, 2012.

[60] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, Aug. 1985.

[61] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proc. ACM ASPLOS*, 2012.

[62] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *Proc. ACM SIGCOMM*, 2011.