CS 695: Virtualization and Cloud Computing

Lecture 10: VM checkpointing and cloning

Mythili Vutukuru

IIT Bombay

Spring 2021

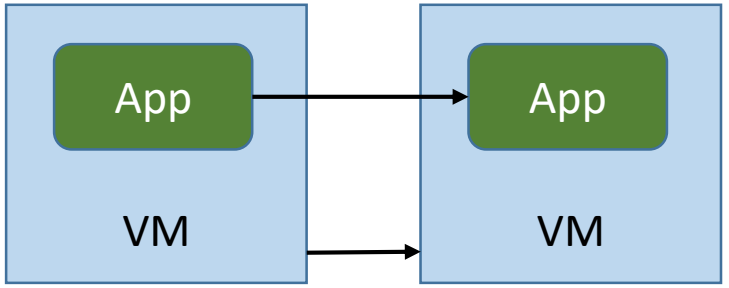
VM checkpointing and cloning

- VM checkpointing for high availability
  - Checkpoint VM memory periodically at backup VM to recover from failures
  - If primary VM fails, backup VM takes over execution using checkpoint state
- VM cloning for parallel execution of tasks
  - Fork a VM with identical state to parent VM to execute tasks in parallel
  - Inspired by idea of forking processes

Remus: High Availability via Asynchronous Virtual Machine Replication
Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield

SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing
H. Andrés Lagar-Cavilla, Joseph A. Whitney, Adin Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, M. Satyanarayanan

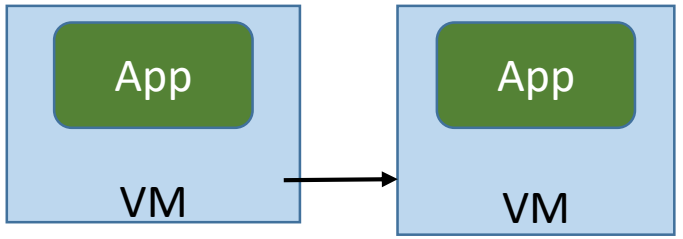
Techniques for reliability



- Application-based replication
  - Application communicates with other replicas and replicates state
  - Consensus protocols like Raft or Paxos used to maintain consistency of replicated state
  - Application decides what state to replicate
  - Reliability via changes to application code (not for legacy apps)
- VM-based replication, or whole system replication
  - Entire VM state (memory, CPU, disk of apps, kernel) is replicated
  - Higher overhead than application-based replication
  - Does not require application code changes (works with legacy apps)
  - VM provides easy way to capture whole system state

Remus: VM replication for high availability

- Primary-backup system, can tolerate single host failures
  - Primary VM runs the application
  - Backup VM gets checkpoint of primary VM state periodically
  - If primary fails, backup resumes from latest checkpoint



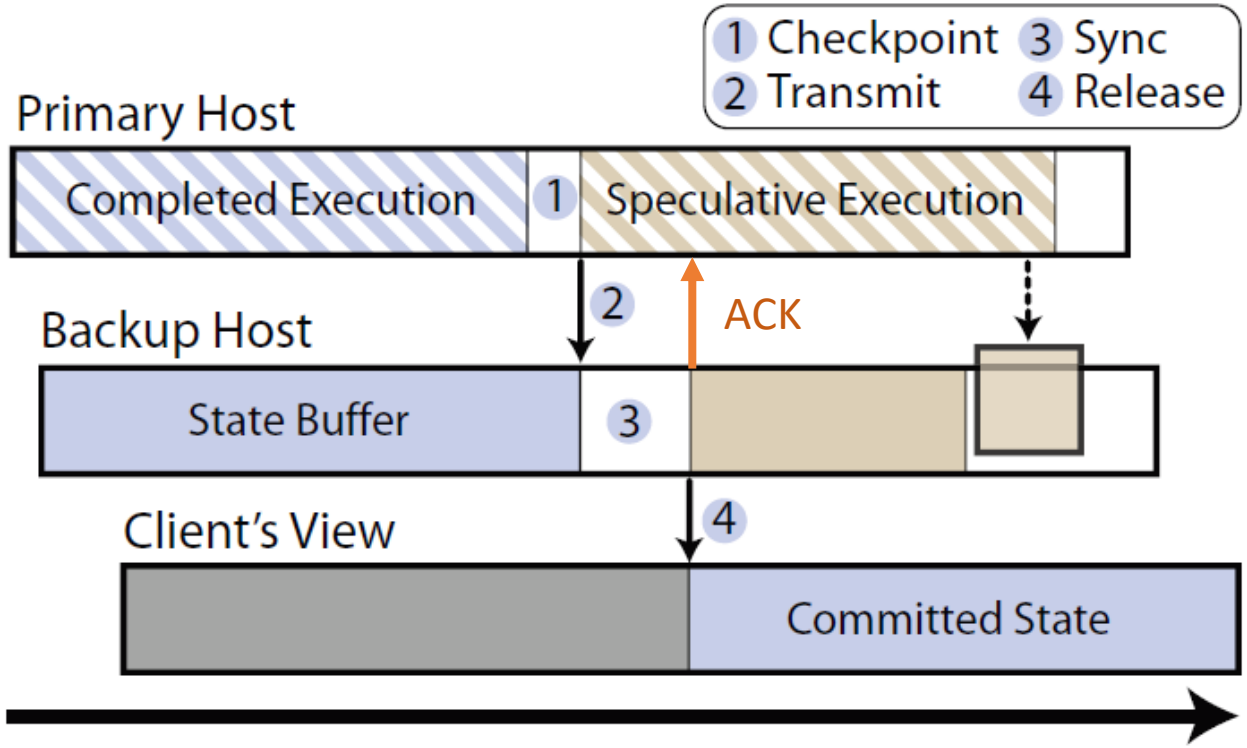
- Periodically (few tens of millisec), primary captures all its state (dirty pages, CPU state, etc.), and transmits it to backup
  - Similarly to iterative pre-copy
- Once backup VM stores this checkpoint, it sends ack back to primary
- While waiting for ack from backup VM:
  - Network output buffered , client will see responses only after checkpoint is done
  - But primary continues execution speculatively, to avoid slowdown
- Asynchronous replication with speculative execution

What to replicate?

- Two ways of replication: replicate state or replay inputs
- For example, a key-value store server processes get/put requests from clients, maintains key-value database
  - Periodically, replicate entire key-value database to backup
  - Or, replay all get/put requests at backup, backup builds copy of database
- Replaying inputs may not always lead to same copy of application state due to non-determinism
  - Random numbers, multicore execution, and so on
- Remus replicates state, does not replay inputs

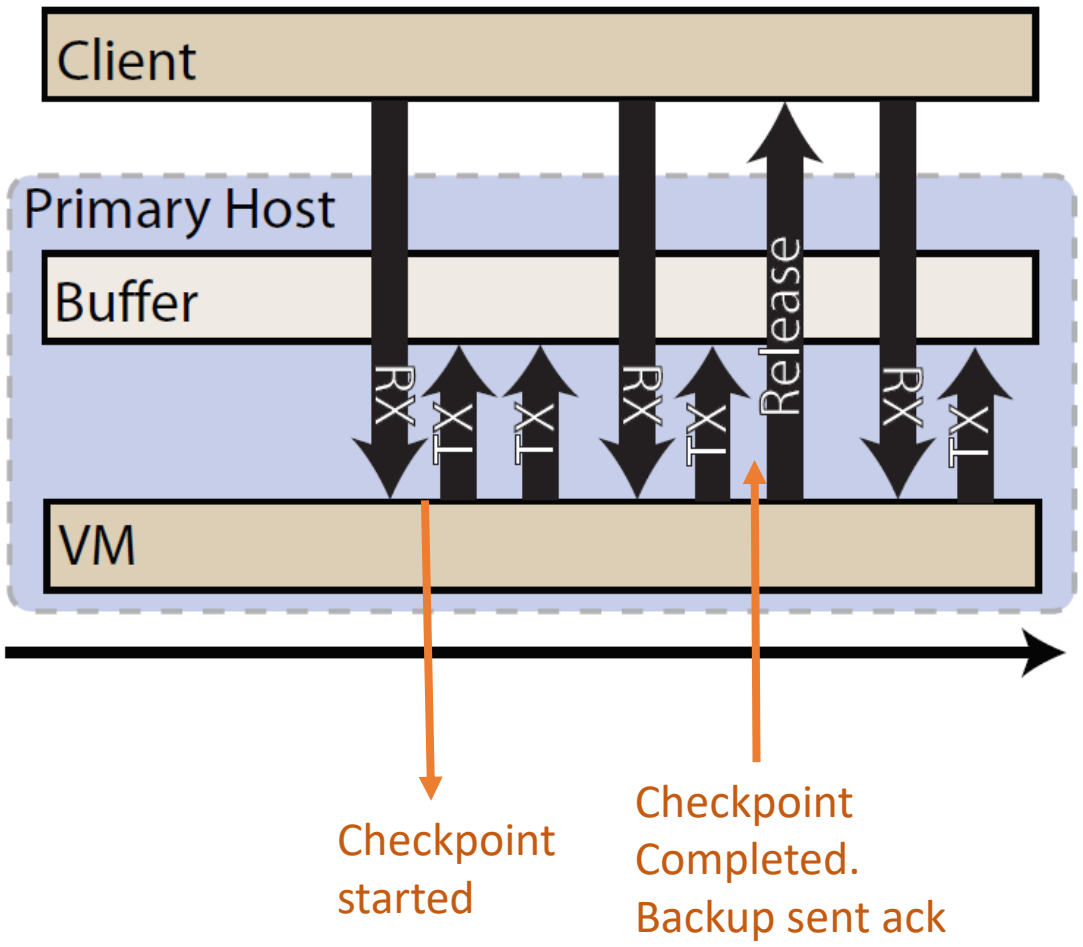
Speculative execution and async replication

1. Once per epoch, pause VM, copy all changed state into a buffer. After state copied, VM resumes speculative execution
   - Shadow page tables of Xen used to track dirty pages in each round
2. Buffered state is copied to memory of backup VM
3. Once all state is received, backup acks the checkpoint
4. Network output released to client from primary



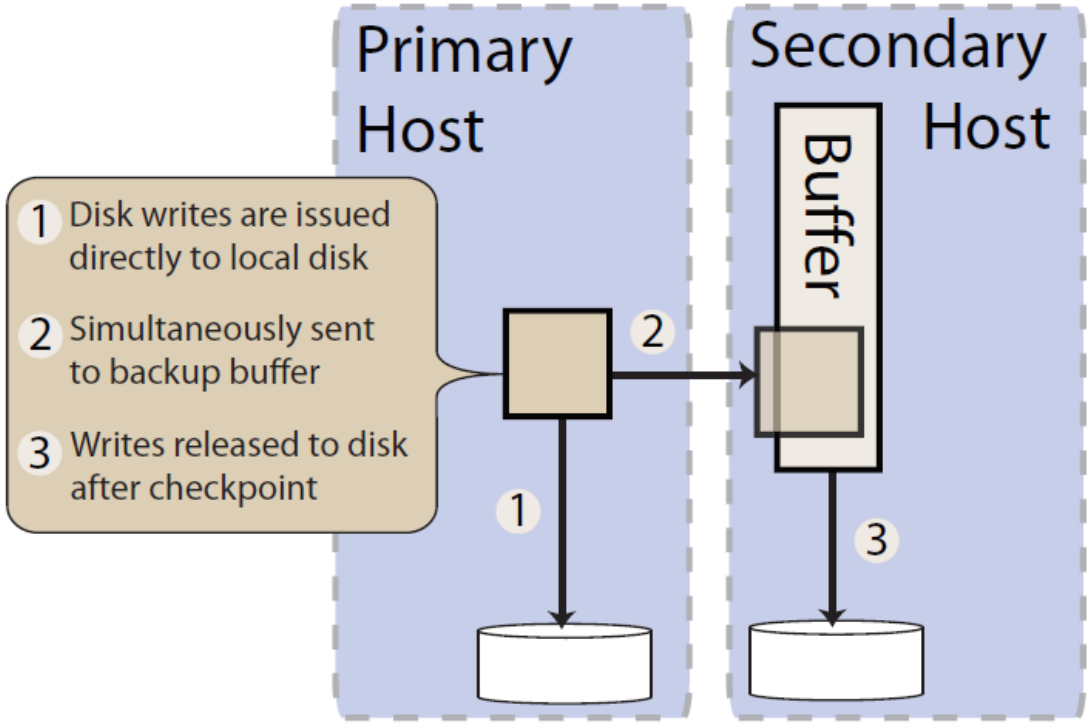
Buffering network output

- Input packets delivered, but output of epoch buffered till checkpoint completes
- Client sees response of epoch after checkpoint completed
- If primary fails after checkpoint completes, backup can continue
  - Ok to release output to client
- If primary fails before checkpoint completes, backup may not be up to date
  - Client won't get response, will retry with backup



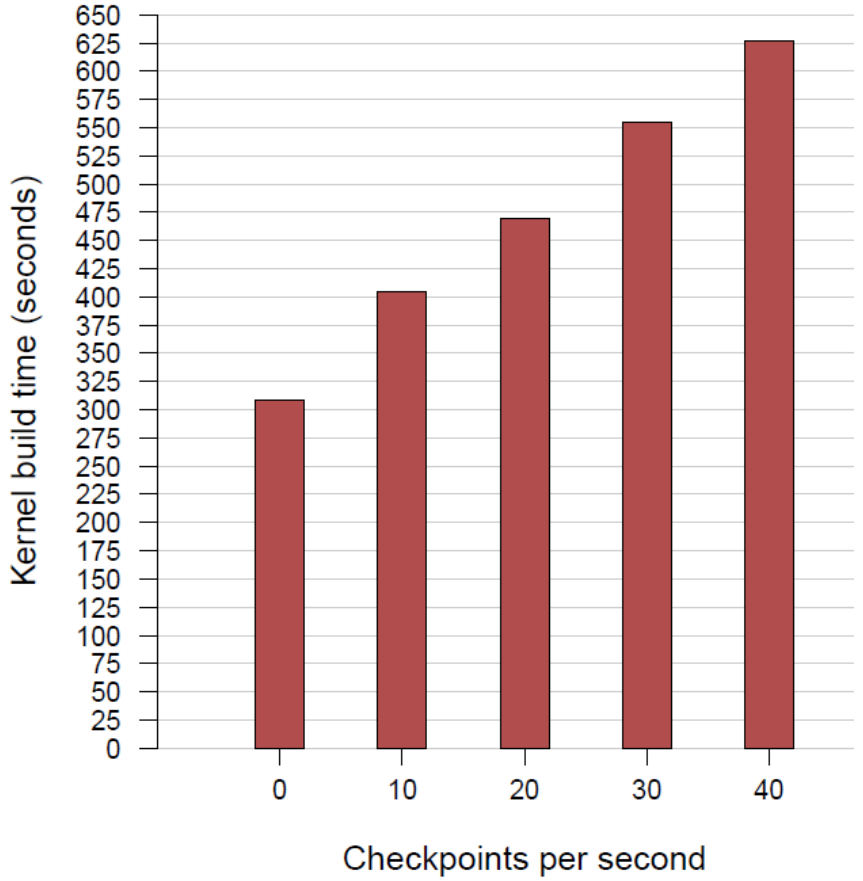
Replicating disk changes

- Disk writes to local disk and sent to backup disk as well
- Write buffers stored in backup memory until checkpoint
  - Do not want the disk to move ahead of memory changes
  - If checkpoint aborted, disk is consistent
- Disk changes flushed to backup disk after checkpoint completes
  - Now, backup ready to take over from primary



Performance overhead of Remus

- Major optimizations to Xen to perform very quick checkpointing of state (~100 microsec) as compared to the original pre-copy migration paper
- Performance overhead ~50% when doing 20 checkpoints/sec
- Slows down applications, especially interactive ones
  - But high availability without app changes



Snowflock: VM cloning via VM fork abstraction

- Advantage of cloud computing: elastically scale number of VMs of an application to match incoming load to servers
  - On-demand scaling of application server replicas
- How to increase the number of VMs running an application?
  - Spawn new VMs when load increases? Takes time to boot up new VM. New VM may not have application state
  - Keep extra VMs idle and ready to take over? Wastage of resources
- Snowflock: implement abstraction of “VM fork”
  - Much like process fork
  - New VM is created quickly, with all state copied from parent
  - Newly forked VM can run in parallel on different host

Example uses of VM fork

(a) Run untrusted code in a sandbox inside forked VM

(b) Run a parallel computation on multiple forked VMs

(c) Fork new VMs to handle extra load, for elastic on-demand scaling

(d) Fork VMs to utilize spare CPU cycles

(a) Sandboxing

```
state = trusted_code()
ID = VM_fork(1)
if ID.isChild():
 untrusted_code(state)
 VM_exit()
else:
 VM_wait(ID)
```

(b) Parallel Computation

```
ID = VM_fork(N)
if ID.isChild():
 parallel_work(data[ID])
 VM_exit()
else:
 VM_wait(ID)
```

(c) Load Handling

```
while(1):
 if load.isHigh():
  ID = VM_fork(1)
  if ID.isChild():
   while(1):
    accept_work()
 elif load.isLow():
  VM_kill(randomID)
```

(d) Opportunistic Job

```
while(1):
 N = available_slots()
 ID = VM_fork(N)
 if ID.isChild():
  work_a_little(data[ID])
  VM_exit()
 VM_wait(ID)
```

VM fork semantics

- Parent VM calls fork, which creates a number of clones or child VMs
  - Only one process inside VM must invoke VM fork
  - Suited for VMs running a single application
- Each child VM is identical with parent, except for different VMID
  - Independent copy, updates at parent not propagated to clone
- Child VMs are on isolated virtual network with parent VM
  - Clone IP address based on VMID, placed in same virtual subnet as parent VM
- Forked VMs run a short-lived computationally intensive job on data slice
- Memory of cloned VM is considered ephemeral, destroyed after VM exits
  - Clone must communicate important data to parent explicitly
- Parent calls join, to wait for all clones to terminate

Alternatives to VM fork

- Suspend a VM, copy its image to multiple VMs, resume from multiple locations
- If copying via NFS, huge network contention at source host
- If copying via multicasting VM image to multiple hosts, performance is better but still takes several minutes
- VM fork aims for under 1 second spawning of forked VMs

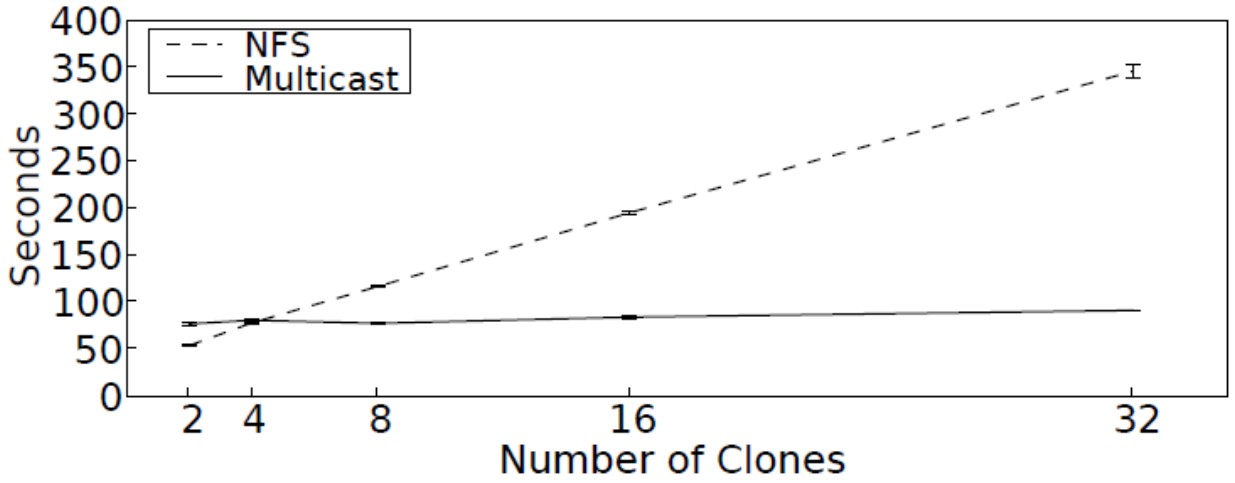


Figure 2: Latency for forking a 1GB VM by suspending and distributing the image over NFS and multicast.

Key ideas in Snowflock

- Upon fork, parent copies VM descriptor to child VM
  - Minimal VM state required to start the clone (VM and device metadata, minimal memory pages, vCPU state, page tables, and so on)
  - I/O devices, memory are minimized before cloning
- After clone starts execution, fetches memory on demand from parent
  - Copy of parent memory at time of fork available at parent VM, fetched by clones on page faults
  - Copy-on-write at parent: Shadow page table at parent. All parent pages marked read only. When parent writes, traps to VMM, copy of parent page is made for parent to modify. Original copy preserved for children
  - Missing page causes page fault at clone: Shadow page table at clone indicates which pages are missing. Xen/domain0 at clone fetches missing page from parent upon page fault.
  - Avoidance mechanisms: avoid fetching memory pages from parent when contents will be rewritten (e.g., I/O buffers)
- IP multicast used to distribute data to clones efficiently (not multiple unicast transfers)
- Disk also modified in parent in copy-on-write manner

Snowflock performance

- Snowflock performs as good as “zero cost fork” (pre-allocated VMs, no cloning or state-fetching overhead)
  - Similar speedup as zero cost fork over single threaded execution at parent

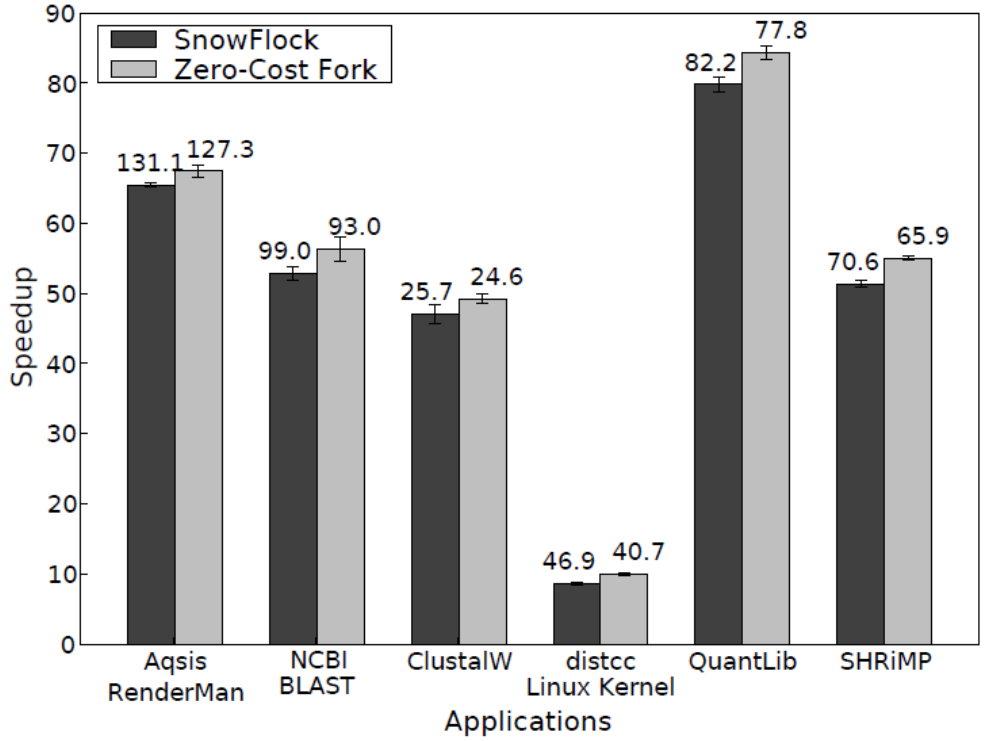


Figure 5: Application Benchmarks. Applications run with 128 threads: 32 VMs × 4 cores. Bars show speedup vs. a single thread zero-cost baseline. Labels show time to completion in seconds.

Summary

- Ideas related to VM migration (pre-copy, post copy)
  - VM checkpointing for high availability
  - VM cloning for parallel performance gains
- VM abstraction makes it easy to capture machine state and migrate/replicate/clone it easily

Remus: High Availability via Asynchronous Virtual Machine Replication
Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield

SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing
H. Andrés Lagar-Cavilla, Joseph A. Whitney, Adin Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, M. Satyanarayanan