# The Case for System Testing with Swift Hierarchical VM Fork

Junji Zhi
*University of Toronto*

Sahil Suneja
*University of Toronto*

Eyal de Lara
*University of Toronto*

## Abstract

System testing is an essential part of software development. Unfortunately, comprehensive testing of large systems is often resource intensive and time-consuming. In this paper, we explore the possibility of leveraging hierarchical virtual machine (VM) fork to optimize system testing in the cloud. Testing using VM fork has the potential to save system configuration effort, obviate the need to run redundant common steps, and reduce disk and memory requirements by sharing resources across test cases. A preliminary experiment that uses VM fork to run a subset of MySQL database test suite shows that the technique reduces VM run time to complete all test cases by 60%.

## 1 Introduction

System testing is an essential part of software development which aims to guard systems against bugs and ensure quality [5]. Yet testing large systems is often difficult. One difficulty lies in configuring a system environment for test [19]. Testers may need to prepare hardware, certain versions of drivers, libraries, packages, or set the applications to certain state. Some non-functional tests, e.g., stress or performance tests, may even require more specialized configurations that are difficult to replicate [10]. In addition, system testing can be time-consuming. For example, "large_tests" in MySQL [16] and Ballista [7] test suites require up to eight hours and 24 hours to execute, respectively.

Cloud computing creates new opportunities for system testing [22, 19, 9]. By configuring a virtual machine (VM) image with the appropriate testing environment, developers can boot multiple instances that leverage the power of the cloud to run a large number of test cases in parallel. This approach leverages the strong isolation properties provided by the VM to formulate testing as an embarrassingly parallel task, where VMs run fully independent test case. However, we observe that this approach requires redundant execution of common test steps, and does not take advantage of the resource sharing opportunities that exists across VMs running different tests.

In this paper, we propose to use hierarchical VM fork to optimize system testing on the cloud. VM fork [14] mimics the semantics of UNIX process fork at VM level. It enables the cloning of a VM into multiple child VMs on the same host or across a cluster. Child VMs inherit the state from their parent including its memory and disk. Since most of the relevant data, executables and configurations needed by a clone are already loaded in its memory by the time of creation, a child VM can readily participate in ongoing tasks such as SIMD computations [2].

The proposed approach has the potential to reduce testing cost on the cloud by eliminating the redundant test steps and expensive clean-up actions. With VM fork, common test steps only get executed once as intermediate state is automatically replicated from parent to child VM(s). Similarly, because different test cases run in isolation, a VM can be simply terminated when a test case completes. In addition, a test framework built on top of VM fork reduces resource requirements by enabling efficient memory and disk sharing between concurrently executing test cases.

To determine the potential benefits of the proposed approach, we conducted a preliminary experiment with a subset of the MySQL test suite. Because the current VM fork implementation does not support hierarchical forking, we used the QEMU snapshots [21] to emulate this functionality. Even in this non-ideal setting (booting a new VM out of a snapshot is slow and VMs do not share memory, only disk), our approach (1) takes 60% less time to complete than an alternative setup that executes test cases sequentially and (2) requires 29% and 70% less CPU cycles and disk space, respectively, than a naive approach that runs test cases in parallel but does not leverage the state sharing opportunities between test

cases.

The rest of this paper is organized as follows: Section 2 discusses the related work in system testing on the cloud. Section 3 introduces our approach and Section 4 discusses the challenges and limitations. Then we present the case study and discuss the preliminary outcome in Section 5.

## 2 System Testing on the Cloud

System testing is essential for ensuring software system quality [5, 8]. It generally consists of three phases: system deployment and configuration, test suite execution, and functional verification or non-functional property measurement. System testing can consume a large portion of software development costs [25].

Each system test case (TC) consists of sequential steps. Table 1 lists some example TC sequences.

Table 1: Test Sequences

| TC# | S1 | S2 | S3 |
|---|---|---|---|
| 1 | A0 | B1 | C1 |
| 2 | A0 | B1 | C2 |
| 3 | A0 | B2 | C3 |
| 4 | A0 | B3 | C4 |

For TC#1, the sequence is A0-B1-C1. The rest TCs are likewise. A TC is considered successful if, after the execution of all its steps, the system behavior conforms to the Functional Requirement Specification(s) [5].

Software testing is starting to migrate to the cloud [18, 19, 22, 15]. System testing on the cloud generally has two execution models: (1) End users request service providers to prepare a pre-configured environment, including OS, web server, database, compiler, testing tools, etc. and submit their systems or applications and the corresponding test tasks. Such model is also named Testing-as-a-service (TaaS) [25, 24], which exposes services like test case auto-generation, and test auto-execution on the cloud, to which end users subscribe; (2) Service providers (e.g., Amazon EC2) provide infrastructure support that allows end users to create independent VMs and optimize the resource consumption for testing activities. One example is D-Cloud [9, 3].

We summarize the benefits of porting software testing to the cloud [12, 22, 19, 9]: (1) cost savings by renting testing infrastructure, (2) accelerated overall test-suite execution in parallel, with each VM packaged with the entire operational environment and encapsulating all dependencies for testing and (3) realistic performance and scalability testing using large cloud infrastructure as compared to limited internal infrastructure. Simple VM checkpointing has also been used for debugging and software testing [23, 17, 6, 20]; however, these previous efforts provide limited sharing between VMs and do not address redundant step execution.

## 3 Our approach

We make the following observations: (1) There exist many commonalities or overlapping steps (e.g., system configuration and test data loading) among TCs. For example, the four TCs in Table 1 all share the same step A0. (2) TCs share the same code base.

To take advantage of the above observations, we propose to use VM fork as a building block to improve system testing efficiency. VM fork is a new abstraction in virtualization-based cloud computing [14]. VM allows rapidly cloning a VM into dozens of replicas running in the same or different hosts. These replicas share the initial state of their parent and thus are stateful workers ready to accept tasks.

According to Observation 1 and 2, since there exist commonalities in test steps and code base, TCs share certain run-time state, including CPU, disk or memory state. VM fork can be leveraged to enable spawning multiple child VMs from a parent on the fly. Each child VM inherits all the ready state of its parent and thus is able to participate in the ongoing testing tasks.

With our approach, TC execution flow changes in two ways: (1) The common steps only execute once and then its resulting state is replicated from parent to child VM(s); (2) VM fork can eliminate the need to execute clean-up steps by simply recollecting or destroying the VMs after the TC execution.

Figure 1 illustrates a workflow instance of executing the TCs in Table 1 with VM fork. Each arrow edge denotes a test step and each cube represents a VM instance. Also, the arrows are tagged with test steps that map to the TCs that we consider in the case study (Section 5).

The workflow begins when the tester configures the VM which hosts the System under Test(SUT). Since all TCs share the common step A0, the base VM is forked into multiple child VMs after executing A0. Similarly, since TC#1 and TC#2 share the common step B1, the VM forks into two instances for these two TCs. To re-use the common steps, a multi-level VM fork mechanism is needed which enables forking a VM multiple times during its lifecycle and thus re-using any sequence of common steps. The more efficient the process of spawning a child VM is, the finer granularity of test step re-use we can achieve. The end result is that VM workers form a hierarchical structure. This is in contrast to starting each VM from scratch and re-doing each configuration or setup step from the beginning.

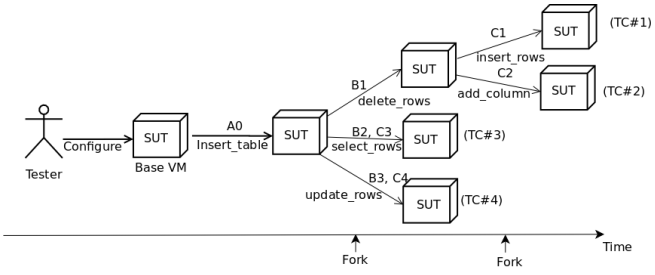


Figure 1: An Example of Testing Workflow with VM Fork

The improvements are two-fold: (1) By reusing common steps among TCs and eliminating the potential clean-up steps, TC execution require less cumulative VM run time; (2) Memory and disk requirement are reduced compared with simply running each TC in a VM.

## 4 Challenges and Limitations

To achieve the benefits of a testing framework based on hierarchical VM fork, we need to find an efficient and practical solution to each of the following challenges:

- Latency and space: The latency of forking VMs is important. We would like to minimize the overhead of forking a new VM so that it does not become a dominant factor in test execution. To do so, a potential approach to optimize VM fork is to implement the memory and disk copy-on-write. Snowflock [14] has shown that fast VM cloning is possible, yet it only implements single-level fork. In terms of space, since VMs are large entities with Gigabytes of footprint, forking multiple VMs can be space-consuming if not done properly. Similar to the strategy that we discussed to reduce latency, disk copy-on-write can be used to reduce space consumption.

- When to fork: The decision of when to fork depends on whether the cost of forking is lower than the benefit of not having to execute the common steps. A run-time profiling may be needed to analyze the cost of each test step.

- Where to fork: Forking locally allows the local memory and storage to be efficiently shared among VMs. However, the increasing number of VMs causes pressure on the local host. When the resource needs exceed the local capacity, it is probably wise to spawn a new VM in a remote host, which achieves scalability and elasticity at the same time. In the latter case, fork latency is influenced by the network bandwidth. The decision of where to fork needs to consider how much forking will happen down the line. Potentially early forks can be used to spread the workload across a large number of hosts and the subsequent ones can be used to leverage the multiple CPUs on each host.

- VM fork transparency: Requiring test engineers to rewrite their test suites to leverage VM fork is unrealistic. Instead, we observe that automatic test suites tend to be written using well-structured test scripts (e.g., Listing 1). We hypothesize that it is possible to leverage static analysis techniques to identify the re-usable common steps and partition test execution automatically.

```
1  #Test 1
2  drop table if exists t1;
3  create table t1;            #config step
4  insert into t1 SOME_ROWS;
5  delete from t1 where CRITERIA1;
6  insert into t1 DELETED_ROWS;
7  drop table t1;            #clean-up step
8
9  #Test 2
10 drop table if exists t1;
11 create table t1;            #config step
12 insert into t1 SOME_ROWS;
13 delete from t1 where CRITERIA1;
14 alter talbe t1 add column C1;
15 drop table t1;            #clean-up step
```

Listing 1: Psuedo-SQL code for two MySQL tests

- Fork side effects: For example, child VMs need to be re-configured so as to avoid IP conflicts with the parent or each other. For VMs that use remote resources (e.g., file systems, or databases), they have to reset their IP connections.

A test framework based on VM-fork is not without some limitations:

- We assume that there exist re-usable overlapping steps among TCs. We anticipate little or no savings if such steps do not exist.

- Hosting an SUT in a VM has an impact on the SUT performance. For some non-functional testing types, such as load testing [1] that requires precise performance measurement, the existence of the virtualization layer may introduce undesired overheads.

## 5 A Case Study: Testing MySQL

We conducted a preliminary case study to provide evidence of the potential benefits of the proposed approach. Because we do not have a working system that supports

hierarchical VM fork, we leverage the existing QEMU snapshot functionality [21] to emulate VM fork. The forking procedure is as follows: For an interim VM (i.e., base VM) that needs to fork into two separate instances, we use KVM to create one snapshot that is identical to the base VM. We then boot another VM (i.e. VM2) based on that snapshot. Since the snapshot uses Redirect-on-Write [11], all the disk changes by VM2 will be preserved in the snapshot delta file (e.g., files with ".qcow" extension) instead of being committed to the base VM image. We can execute a TC in VM2 and then discard the snapshot to prevent that TC from polluting the base VM. This approach, however, has two limitations: (1) booting a new VM out of a snapshot is slow and (2) VMs do not share memory, only disk. Therefore, our results are conservative.

## 5.1 System under Test

We experiment with the MySQL(v5.5) "large_tests" suite, which executes the common database operations on a set of large tables with 274 trillion rows. For simplicity, we use a smaller test table (with 1.1 trillion rows) in our study. Nevertheless, constructing such a large test table for each TC is expensive in time and space. Whereas the complete test suite includes tens of TCs, we focus our effort on a small subset of 4 representative TCs that exercise database insert, update, delete, alter, select and update functions. The TCs are shown in Table 2.

Table 2: MySQL Test Cases

| TC# | Description |
|---|---|
| 1 | Construct a large table, delete a large number of rows which conform to a certain criteria, and then inserting the deleted rows. |
| 2 | Construct a large table, delete a large number of rows which conform to a certain criteria, and then add a new column to the table. |
| 3 | Construct a large table, select a large number of rows which conform to a certain criteria. |
| 4 | Construct a large table, update a large number of rows which conform to a certain criteria. |

## 5.2 Experiment

We run the four TCs in three configurations. The baseline runs each TC in sequential order in a single host (Config. I), which is the existing approach of executing the MySQL TCs. In Config. II, we run each TC in a separate VM concurrently. In Config. III, we run TCs with the emulated VM fork. In order to re-use the test steps, we first identify one common step among four TCs, i.e., constructing a large table (insert table), and a common step (i.e., delete table) between TC#1 and TC#2. Therefore, forking a VM after these common steps is our way to re-use the steps. Figure 1 presents the test suite execution workflow.

## 5.3 Hardware Setup

Our hardware setup is as follow: the MySQL (v5.5) SUT is hosted in a 64-bit Ubuntu 12.04 (Kernel v3.2.0) VM. The size of VM image is 21GB. The KVM hypervisor (v3.8.0) [13] operates on an Ubuntu 12.04 (Kernel v3.8.0) host OS and a QEMU emulator (v1.0). We also use libvirt(v1.0.4), a VM management user interface utility, in our experiments. The host OS runs on a physical machine with a 64-bit 4-Core AMD A10-5700 APU processor, 12GB RAM and a 2TB hard drive.

## 5.4 Experiment Outcome

We run the experiment three times and average the statistics. In Figure 2, the first bar presents the outcome of running in Config. I and the rest four bars denote Config. II. Since each TC shares the first step (i.e., insert table), all bars share a segment of almost the same length (coloured blue in Figure 2). The blue segment also repeats itself at the first bar since other three TCs needs to re-execute this step again.

Figure 3 presents the outcome for Config. III. Compared with Config. I and II, there are two additional types of cost incurred by the emulated VM fork: the overhead of creating VM snapshots and booting a new VM from a snapshot. During the execution of TC#1, its host VM is checkpointed twice to create two snapshots which are reused for other TCs. The checkpointing costs are denoted by two tiny green segments at the bottom bar in Figure 3. The cost of booting VMs from snapshot is denoted by the brown segments of the other three bars.

We also measure the VM snapshot size for each TC and compare them with the base VM size in Config. I and II. The result is shown in Figure 4.

We make the following observations:

1. Running TCs in parallel with the emulated VM fork reduces execution time and its performance is only limited by the longest TC (i.e., the bottleneck) in the test suite. More concretely, the bottleneck in Config. III is TC#1, which takes 401s while executing TCs sequentially in Config. I consumes 971s. Thus Config. III consumes 60% less the time of the sequential execution

2. Config. III requires less resources to run than Config. II. The savings come from the fact that there is

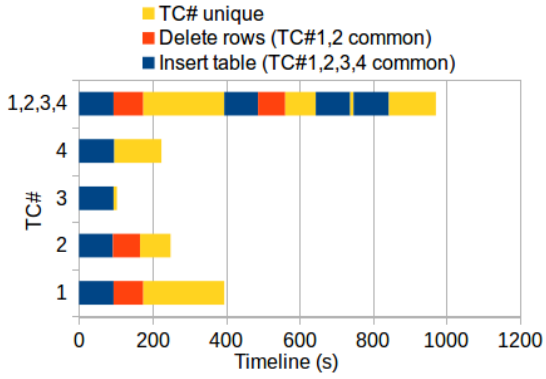


Figure 2: Serial (Config. I) and Naive Parallel (Config. II) Runtime.

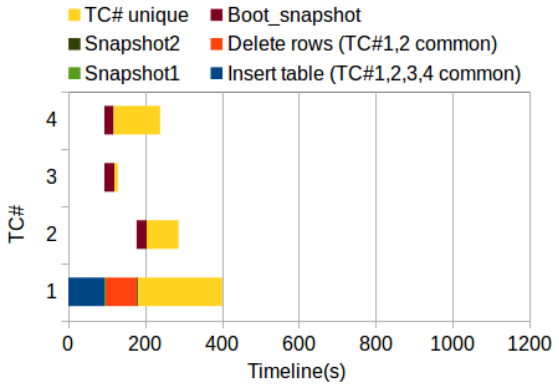


Figure 3: Hierarchical VM Fork (Config. III) Runtime.

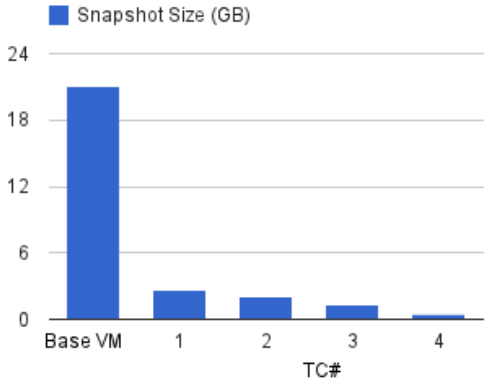


Figure 4: Disk Usage of Snapshots

no need to execute the common steps (i.e., blue segments in VM #2, #3 and #4 and orange segments in VM #2) in Config. III. Also, the cost of creating snapshots in TC#1 is negligible. As a result, testers do not have to pay for the run time associated with re-executing these test steps. This is shown in the cumulative execution time of TCs: Config. III takes 690s in total while Config. II consumes 971s (29% savings).

3. Creating a snapshot of the base VM for a TC consumes less than 5% disk space of the base VM image(Figure 4). Therefore, Config. III requires only one base VM image and four snapshots while Config. II requires four base VMs (70%+ saving).

4. The cost of the test environment configuration step, which is not shown in the figures, are saved in Config. III, since any child VMs forked from a parent VM will automatically inherit all its state. Although the process can be automated (e.g., by Shell scripts) to prepare each VM for test in Config. II, it requires some engineering efforts to compose and manage such automation tools or scripts, which is not needed with VM fork.

## 6 Conclusion

In this paper, we propose to leverage the technique of hierarchical VM fork to optimize system testing in a virtualization-based cloud. Testing on the cloud can yield the advantage of speeding up TCs by running in parallel. The proposed approach reduces the cost of running system tests on the cloud by eliminating the redundant steps among TCs and by allowing VMs to share memory and disk.

Our preliminary experiment uses an emulated hierarchical VM fork implementation to execute a small portion of the MySQL test suite. Even in this limited setup, our technique reduces the test completion time by 60% compared to the serial execution, and lowers the CPU and disk space usage by 29% and 70%, respectively, compared to a naive approach that runs TCs in parallel but does not leverage the state sharing opportunities between TCs.

For future work, we plan to implement the discussed hierarchical VM fork. We envision a test framework which incorporates VM fork and facilitates system testing on the cloud.

## References

[1] AVRITZER, A., AND WEYUKER, E. J. Generating test suites for software load testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis* (1994), ACM, pp. 44–57.

[2] BALMER, K., GOVE, R. J., GUTTAG, K. M., AND INGSIMMONS, N. K. Multi-processor reconfigurable in single instruction multiple data (simd) and multiple instruction multiple data (mimd) modes and method of operation, May 18 1993. US Patent 5,212,777.

[3] BANZAI, T., KOIZUMI, H., KANBAYASHI, R., IMADA, T., HANAWA, T., AND SATO, M. D-cloud: Design of a soft-

ware testing environment for reliable distributed systems using cloud computing technology. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), IEEE Computer Society, pp. 631–636.

[4] BERGMANN, S. Phpunit fixtures. `http://phpunit.de/manual/current/en/fixtures.html`, 2014. [Online; Accessed: 2014-05-23].

[5] BRIAND, L., AND LABICHE, Y. A uml-based approach to system testing. In *ł UML 2001The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Springer, 2001, pp. 194–208.

[6] CLOUDSTACK. Devcloud continuous tests. `https://cwiki.apache.org/confluence/display/CLOUDSTACK/Devcloud+Continuous+Tests`, 2014. [Online; Accessed: 2014-03-01].

[7] CMU. Ballista os robustness test suite. `http://users.ece.cmu.edu/~koopman/ballista/ostest/ostest_download.html`, 2014. [Online; Accessed: 2014-02-10].

[8] GAROUSI, V., AND ZHI, J. A survey of software testing practices in canada. *Journal of Systems and Software 86*, 5 (2013), 1354–1376.

[9] HANAWA, T., BANZAI, T., KOIZUMI, H., KANBAYASHI, R., IMADA, T., AND SATO, M. Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on* (2010), IEEE, pp. 428–433.

[10] HEDGES, R., LOEWE, B., MCLARTY, T., AND MORRONE, C. Parallel file system testing for the lunatic fringe: The care and feeding of restless i/o power users. In *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on* (2005), IEEE, pp. 3–17.

[11] IBM. Ibm developerworks. `http://www.ibm.com/developerworks/tivoli/library/t-snaptsm1/index.html`, 2014. [Online; Accessed: 2014-02-10].

[12] INÇKI, K., ARI, I., AND SOZER, H. A survey of software testing in the cloud. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on* (2012), IEEE, pp. 18–23.

[13] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[14] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), ACM, pp. 1–12.

[15] LIU, H., AND ORBAN, D. Remote network labs: An on-demand network cloud for configuration testing. *ACM SIGCOMM Computer Communication Review 40*, 1 (2010), 83–91.

[16] MYSQL. Mysql5.5 manual. `https://dev.mysql.com/doc/refman/5.5/en/mysql-test-suite.html`, 2014. [Online; Accessed: 2014-02-16].

[17] ORACLE. Oracle enterprise manager. `https://blogs.oracle.com/oem/entry/snap_clone_instant_database_on`, 2014. [Online; Accessed: 2014-03-01].

[18] ORSO, A., AND ROTHERMEL, G. Software testing: A research travelogue (2000–2014). In *Future of Software Engineering Session (FOSE'14), 36th International Conference on Software Engineering (ICSE'14)* (2014).

[19] PARVEEN, T., AND TILLEY, S. When to migrate software testing to the cloud? In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on* (2010), IEEE, pp. 424–427.

[20] PENG, F., DENG, B., AND QI, C. Caste: a cloud-based automatic software test environment. *World Academy of Science, Engineering and Technology 71* (2012), 2012.

[21] QEMU. Qemu snapshot. `http://wiki.qemu-project.org/Documentation/CreateSnapshot`, 2014. [Online; Accessed: 2014-02-25].

[22] RIUNGU-KALLIOSAARI, L., TAIPALE, O., AND SMOLANDER, K. Testing in the cloud: Exploring the practice. *Software, IEEE 29*, 2 (2012), 46–51.

[23] VMWARE. Vmware support. `https://www.vmware.com/support/ws55/doc/ws_clone_overview.html`, 2014. [Online; Accessed: 2014-03-01].

[24] YU, L., LI, X., AND LI, Z. Testing tasks management in testing cloud environment. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual* (2011), IEEE, pp. 76–85.

[25] YU, L., TSAI, W.-T., CHEN, X., LIU, L., ZHAO, Y., TANG, L., AND ZHAO, W. Testing as a service over cloud. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on* (2010), Ieee, pp. 181–188.