A Little Language for Rapidly Constructing Automated Performance Tests

Shaun Dunning
NetApp, Inc.
Kit Creek Road P.O. Box 13917
Research Triangle Park, NC 27709, USA
shaun.dunning@netapp.com

Darren Sawyer
NetApp, Inc.
475 East Java Drive
Sunnyvale, CA 94089, USA
darren.sawyer@netapp.com

ABSTRACT

In order to effectively measure the performance of large scale data management solutions at NetApp, we use a fully automated infrastructure to execute end-to-end system performance tests. Both the software and user requirements of this infrastructure are complex: the system under test runs a multi-protocol, highly specialized operating system and the infrastructure serves a diverse audience of developers, analysts, and field engineers (including both sales and support). In this paper we describe our approach to rapidly constructing automated performance system tests by using a lightweight, little, or domain-specific language called *SLSL* in order to more effectively *express* test specifications.

Using a real world example, we illustrate the efficacy of *SLSL* in terms of its *expressiveness*, *flexibility*, and *ease of use* by showing a complex test configuration expressed with just a few language constructs. We also demonstrate how *SLSL* can be used in conjunction with our performance measurement lab to quickly deploy performance tests that yield highly *repeatable* measurements.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; D.3.2 [**Software**]: Programming Languages—*Language Classifications*

General Terms

Language, Measurement, Performance

Keywords

performance, test automation, domain-specific language, little language

1. INTRODUCTION

Emerging technologies like virtualization and cloud computing are having a profound impact on the ways that data storage systems are configured and deployed in modern data centers. In the past it would be common to see data from a relatively small number of different applications hosted on a single storage system. Today we are seeing unprecedented levels of consolidation, resulting in data from many applications, possibly from many different departments within a company or even from multiple companies hosted on a single shared storage system. Enterprise storage systems from companies like NetApp are evolving their storage architectures to meet the challenges of today's data centers. While automated storage management tools may simplify the processes used by a storage administrator to configure and provision storage in these environments, the level of complexity of the resultant system configurations can be quite high. This complexity, coupled with the increased levels of storage consolidation, greatly increases the demand on both the scale and flexibility of the performance measurement tools and processes used by engineering to thoroughly test and characterize the performance of our systems. The impact of the changes needed to fulfill the growing demands on performance testing posit the need for the ability to quickly and reliably build and deploy new, automated performance tests. This paper describes a simple language-based approach to rapidly create complex automated storage performance test scenarios that are representative of the environments in which we increasingly find our systems being deployed.

The organization of the rest of this paper is as follows: Section 2 provides background information on performance testing of storage systems, the performance testing environment at NetApp, and the structure of NetApp storage systems. The concepts presented should also serve to motivate our approach detailed in Section 3, where we introduce a *little language* for describing the setup and execution of an automated storage performance test. Each subsection of Section 3 details a different test automation phase and includes mini code examples that are used to demonstrate how the language can be applied to meet the objectives of that phase. The section concludes with a description of the Python-based implementation of the little language. Section 4 follows with a more extensive example that demonstrates how the language can be used to create a rather complicated test scenario with a reasonably small amount of code. Section 5 summarizes the conclusions of this work and includes proposals for future extensions.

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

2. BACKGROUND AND MOTIVATION

2.1 Storage System Performance Testing

Evaluating the performance of a storage system typically involves measuring its behavior while servicing a series of I/O requests. The I/O requests originate from one or more "load generator" programs that run on general purpose compute server hardware that has access to a storage system through one or more standard storage interfaces. We refer to the hardware running the load generator programs as *clients* of the storage system, though *hosts* is another commonly used term. The storage system under test may be directly attached to a client (typically referred to as direct-attached storage, or *DAS*) or may be connected via some form of storage network, over which the client communicates using one or more network storage protocols. Network storage protocols allow a client to access data on the storage system by making requests related to specific blocks of data over a Storage Area Network (*SAN*) protocol (e.g., FCP or iSCSI), or to specific files made available by the storage system over a Network Attached Storage (*NAS*) protocol (e.g., NFS or SMB). The diagram in Figure 1 shows the physical layout of a 4 node storage system where the load generating clients send I/O requests using one of the network storage protocols through a data network switch that connects the clients to the storage.

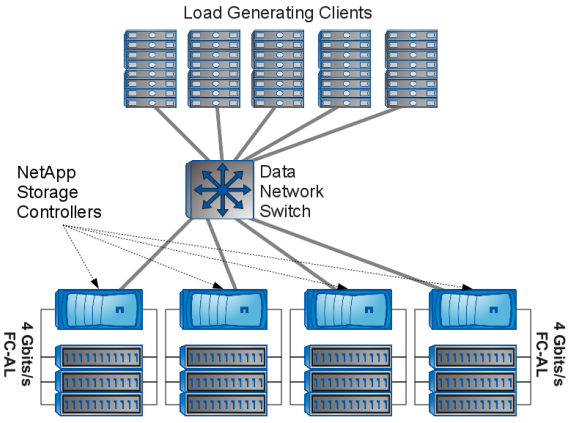

Figure 1: Physical View of a System Under Test

The characteristics of the series of I/O requests made by a load generator to the storage system defines the *workload* that the storage system must service. The workload may be an actual application making I/O requests in support of functions it is providing to a user, or it may be a simulated workload designed to mimic the I/O request stream of a real application or to exercise specific sets of operations of interest during the evaluation. A client machine may execute one or more load generator programs, each of which may create one or more workloads to be served by the storage system under test. A workload may also originate on the storage system itself. This could be an internal process that operates on the data resident on the storage system (e.g., data verification, virus scanning, data deduplication) or that sends data to another storage device, typically for data replication or protection (e.g., tape backup, data mirroring to a disaster recovery storage device, etc. . .). While some of these processes may run continually on a storage system, others may need to be explicitly initiated at a specific point of interest during the performance test. Observing how this "background" load created by *internal workloads* affect the service delivered to "foreground" workloads originating from external load generators is particularly interesting in the evaluation of storage systems as the tasks performed by internal background workloads are common, and important, to the proper operation of a storage infrastructure.

Evaluation of the storage system is performed by collecting metrics of interest over the course of the time the workload is being serviced. Metrics may be observed from either the load generator itself or from the storage system under test. Load generators typically report the amount of work, or *throughput*, delivered by the storage system over some period of time, and the *latency*, or response time, of requests made to the storage server. Both types of metrics may be expressed in storage-centric terms (e.g., megabytes/second, I/Os per second, microseconds per I/O) or in terms related to higher level transactions that result in a series of I/O operations to the storage device. Metrics collected on the storage server may provide the combined, storage-centric view of the throughput and latency from all load generators accessing the storage, as well as storage server resource utilization metrics for various components within the storage server (e.g., CPU utilizations, utilization and response times for individual disk devices, etc. . .). *Measuring* key metrics on both the load generators and storage server provides a comprehensive view of the performance of the storage system.

It is important to note that the storage system under test may not consist of a single storage server, but rather may be a network of storage servers that together provide clients access to various pieces of data. Access to any particular piece of data may be provided through all of the constituent servers or through some subset. Thus there may be different paths by which data may be accessed, and not all paths may be able to reach any given pool of data created within the subsystem. Specifying how load is presented to such systems during a performance test can be particularly complex, but is useful in evaluating how using different combinations of paths through a network of storage servers can affect performance delivered to load generators.

2.2 Performance Measurement at NetApp

Storage system performance testing at NetApp serves the widely varying needs of numerous internal customers. During design and development, performance analysts and software developers execute tests to evaluate the performance of new features or enhancements to existing features. Performance regression tracking tests are run continually during this phase on a wide range of configurations and workloads to identify unexpected and, not surprisingly, usually negative changes in performance. Before the release of a new hardware or software product, a final set of characterization measurements is performed to provide information that is used by our sales and support organizations. Also during this time industry standard performance benchmark results are generated and publicly published. Finally, customer sales or support engagements occasionally require reproducing a customer workload using a customer script or program, or using a simulated workload with one of many possible I/O load generating tools.

The demands placed on our performance test infrastructure require not only the capability to handle a large volume of performance tests, but *extreme* flexibility in the types of supported tests and configurations. Additionally, there is a high degree of variance in performance measurement expertise amongst the users who wish to run performance tests. We need to provide a simple interface for the average software developer who just wants to see if proposed code changes affect system performance, while providing the ability for performance analysts who want to collect comprehensive performance data on a complex set of configurations and workloads.

To meet these demands, NetApp invested heavily in the creation of a completely automated performance test environment. Human intervention required for any performance test is limited, in almost all cases, to providing a test specification to our automated performance test system. The configuration of physical components used during a test, including load generating clients and storage servers, as well as the physical connections between them, is performed with automated scripts that reserve and connect required resources out of pools of clients, storage servers, and disk drives. Physical equipment reconfiguration is generally only required when resources are added or removed from the pools. Once the physical test configuration is completed, configuration of operating system parameters on both the client machines and the storage server is performed, as is creation of disk pools, logical storage objects located with them, and configuration of storage network ports. It is the automated process of configuring the test environment once the physical equipment is in place that is the primary focus of this paper.

This high degree of automation allows us to complete many times more performance tests than we would be capable of with manual execution. It ensures tests are run in an extremely *repeatable* manner, removing almost any chance of test execution flaws that can result during manual setup of complex test environments. Anyone in the company who needs to run a performance test can do so with minimal training, while those with more extensive testing needs are given the flexibility to meet their objectives using the same test environment. The remaining sections of this paper explain the core software infrastructure that allows us to meet the extensive demands facing our performance testing environment.

2.3 NetApp Storage and Data ONTAP®

At the core of all NetApp storage systems is the unifying Data ONTAP®software technology. Data ONTAP®runs on the full array of NetApp storage controller hardware to provide shared access to stored data. Data can be simultaneously accessed through any of the most common block-based and file-based networked storage protocols, including NFS, SMB, iSCSI, and FCP. In this section, we'll describe the general structure of NetApp storage platforms and the Data ONTAP®architecture as it relates to the methods we use later to describe the systems that we want to target with automated performance tests.

The physical structure of a NetApp storage system consists of one or more storage controllers that contain CPUs to process requests, memory used for caching data, and I/O controller expansion slots. Each controller provides a number of "front-end" network ports to connect to external client machines that make storage requests over Ethernet or fiber channel networks. It also provides storage ports to connect to physical disk drives via a storage interconnect (e.g. fiber channel arbitrated loop or SAS). Disk drives connected to a storage controller can be divided into groups called *aggregates*. Disks within an aggregate are arranged into RAID groups that provide a selection of protection levels against single or double disk failures. When configured in *cluster-mode*, two or more storage controllers can be connected via a cluster network that allows requests entering on a network port of one controller to access data on an aggregate attached to another controller.

NetApp uses virtualization technology to build a logical view of the storage system on top of the physical controllers and disk storage devices. A system administrator can construct data *volumes*, one or more of which can reside on top of a physical disk *aggregate*. The volume is then further divided into LUNs or files depending on the type of storage protocol used to access the data. Physical network ports can host one or more "logical interfaces", or *LIFs*. LIFs are assigned a particular address, IP address for Ethernet ports or a WWN address for fiber channel network ports, and are used to export data in a volume or a LUN to clients on a network. Any subset of volumes and LIFs can be grouped together to form a secure "virtual server", or *vserver*, within the physical storage server. Much like virtual machines that can be hosted on a physical compute server, storage vservers appear as completely separate storage systems to the outside world, while many vservers may actually share underlying physical storage hardware [2]. Figure 2 shows how storage vservers span physical hardware boundaries. In this example clients access volume data through LIFs hosted on one of three storage vservers and are unaware of the four underlying physical storage controllers.

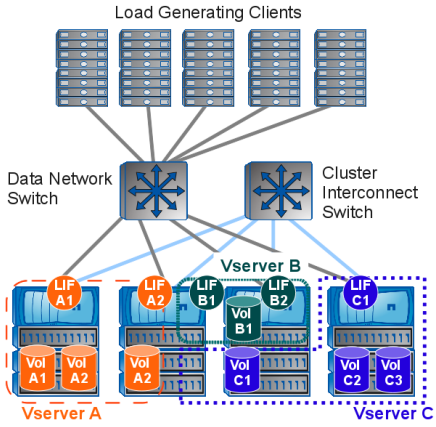

Figure 2: Logical View of a System Under Test

Virtualized storage objects are less constrained than the physical objects on which they reside. A physical storage server may play host to hundreds or thousands of vservers, each with hundreds or thousands of volumes and LIFs. In fact, such configurations are becoming more and more common as storage administrators and cloud storage service providers consolidate data from many different applications from different users or "tenants" [3].

Needless to say, attempting to create test scenarios in

which a storage system is simultaneously serving data to many different tenants, each with its unique set of applications running on clients accessing many volumes of data over any of the various storage protocols, can be a particularly daunting task. Yet it is the type of world we wish to be able to describe and automatically build with the automated performance test system described in the rest of this paper.

3. LITTLE LANGUAGE APPROACH

SLSL (pronounced "*sizzle*") is designed to provide flexibility in the construction and configuration of performance tests for data storage systems. We highlight these benefits in flexibility within the context of four automation *phases* of performance testing shown in Figure 3. Automation tasks in the *Hardware Configuration* phase involve the configuration of *physical* testbed components. Because of the static nature of these components, there is less opportunity for SLSL to enhance the richness of supported configurations in this phase of automation. Conversely, automation tasks associated with the configuration of *logical* test components, represented in Figure 3 by phases `2`, `3`, and `4`, are much more flexible in nature and allow for a greater number of potential valid configurations. Therefore, SLSL is mainly focused on providing high-level abstractions that support of a rich variety of *logical component* configurations.

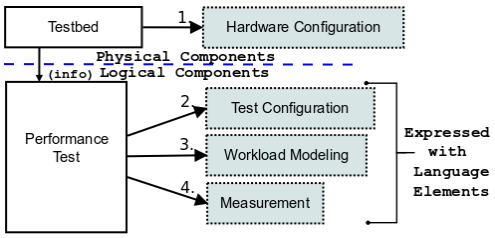

Figure 3: Phases of Performance Testing

The philosophy behind our approach is to identify common *tasks* within each automation phase and provide succinct, high-level language abstractions to encapsulate these tasks. SLSL abstractions can represent a trivial task, like sending a message to a testbed component, or a complex task, like executing a collection of load generation processes in parallel from multiple hosts to multiple storage servers. In this way these tasks, no matter how trivial or complex, become *reusable units* that can be *specified* using language elements. The re-usability of SLSL elements implies they can be "mixed-and-matched" in support of a wide variety of possible performance test implementations.

SLSL has three key language properties: ***(i)*** it is a *declarative* language [7], ***(ii)*** it is *hierarchical*, and ***(iii)*** it is a *little*, or *domain-specific*, language [20, 8]. SLSL declarations are typed using a reserved word `type` parameter, while the remaining parameters are tailored specifically to the `type` of element being declared. This technique allows language elements to be *overloaded* such that a single element can abstract multiple, similar, tasks. Complete SLSL specifications result in a hierarchical relationship between declarations because certain SLSL elements are used to declare other SLSL elements. Figure 4 illustrates these concepts using two `client` declarations, of type "`linux`" and "`windows`", that are later used to declare `ports` on lines 3 and 4.

```
1 c1 = client(type="linux")
2 c2 = client(type="windows")
3 port(type="ethernet", client=c1, name="eth1")
4 port(type="fc_initiator", client=c2, name="1a")
```

Figure 4: SLSL `port` and `client` Declarations

Each automation phase in Figure 3 has an associated collection of SLSL elements that target the common tasks performed within each phase. Table 1 groups SLSL elements according to their associated phase and shows the various types of declarations supported by each element.

Language Element	Supported `types`
Hardware Configuration Phase	
`controller`	*default*
`client`	linux, windows
`port`	ethernet, fc_target, fc_initiator
`aggregate`	*default*
`cluster`	*default*
Test Configuration Phase	
`vserver`	*default*
`volume`	7mode, clustered
`datastore`	volume, LUN
`lif`	tcp_ip, fcp
`io_path`	nfsv3, nfsv4, fcp, smb
Workload Modeling Phase	
`path_mapper`	round_robin, linux_multipath
`workload`	sio, spc1, sfs, iozone, iometer, filebench
Measurement Phase	
`command`	bsd, dblade, ngsh, linux, windows
`event`	timer
`result_collector`	perfstat, perfant
`iteration`	*default*
`test_runner`	*default*

Table 1: SLSL Elements Grouped by Phase

3.1 Hardware Configuration Phase

The first phase of automation involves configuring the testbed hardware components. This includes tasks like booting, installing operating systems, and configuring network switches. There is little opportunity for SLSL to enhance flexibility at this phase due to the static nature of these configurations and it is required that this phase be completed before SLSL can be applied. In our performance measurement lab discussed in Subsection 2.2, we use hardware configuration automation to build a testbed on-demand and then populate designated SLSL structures with hardware configuration information as shown in Figure 5 with an example declaration of a gigabit ethernet `port e0c` residing on a storage `controller fas6080_1`. Other static hardware information is captured similarly using the remaining *Hardware Configuration* SLSL elements shown in Table 1.

```
1 n1 = controller(mgmt_ip="192.168.2.1",
2                 hostname="fas6080_1")
3 port(type="ethernet", name="e0c", node=n1)
```

Figure 5: Ethernet `port` on a Storage `controller`

3.2 Test Configuration Phase

The second phase of automation shown in Figure 3 involves configuring the *logical components* of the test. We define two high-level automation tasks in this phase: ***(i)*** configuring datasets, and ***(ii)*** specifying how datasets can be accessed. Each task contains key *properties* that affect the behavior of the system under test and, ultimately, the outcome of the results. Therefore, these properties are made accessible through SLSL elements so that they can be configured by the test designer.

3.2.1 Dataset Configuration

The first task of the *Test Configuration* phase is configuring the dataset. In our context, we define the *dataset* as the total set of data objects from which a subset will be chosen as the target for the measurable work. To effectively abstract the task of dataset configuration, SLSL needs to express the following key properties:

- **size**: how big is the dataset?
- **type**: what type of data is being stored? (*e.g.,* files, LUNs, etc. . .)
- **attributes**: are there any "special" attributes of the datastore? (*e.g.*, de-duplicated, compressed, etc. . .)
- **container**: where and how does the dataset reside on the storage? (*e.g.*, *aggregates* and *volumes* in Data ONTAP®)

Table 2 highlights how the configuration of these properties is useful in the context of data storage system performance testing.

Property	Uses in Performance Testing
size	control access at various cache levels
type	target specific storage sub-systems
attributes	test specific data management features
container	control over various container properties
	control where dataset gets stored

Table 2: Effects of Dataset Properties on Storage System Performance

SLSL provides a `datastore` element representing a collection of data objects related by *size*, *type*, *attributes*, and *container*. Figure 6 shows an example of a volume datastore containing 100 files, each 10 gigabytes in size, residing in a 16 terabyte Data ONTAP®volume container.

```
1 vol1 = volume(size="16t", name="vol1",
2               server=n1)
3 datastore(type="volume", container=vol1,
4           num_files=100, file_size="10g")
```

Figure 6: SLSL `datastore` and `volume` Declarations

3.2.2 Dataset Access Configuration

The second task in the *Test Configuration* phase is defining the set of paths in the system that *can* be used to access the `datastores`. This is not to be confused with the configuration of the paths that *will* be used to access `datastores` that is later described in Susbsection 3.3. The key properties of dataset access that need to be specified with SLSL elements are:

- **end points**: what are the end points from which data access requests are transmitted and received?
- **available data**: what datastores are exported through paths defined?
- **protocol**: how do end points communicate with one another?

Table 3 highlights how these properties are useful in controlling various aspects of performance testing.

Property	Uses in Performance Testing
end points	control physical capabilities
	test bandwidth/limits of inter-link
	identify bottlenecks in inter-link
available data	control optimal and sub-optimal paths
protocol	control of protocol-specific properties
	test protocol limits and overhead

Table 3: Effects of Data Access Properties on Storage System Performance

The SLSL elements `lif` (abbreviation for *logical interface*) and `io_path` are designed to allow for the configuration of dataset access. `lif` abstractions represent end points, while `io_path` abstractions encapsulate a single *path* from which I/O requests can travel between two end points to the data. Figure 7 shows these concepts using two storage server TCP/IP `lifs` declared on lines 1–6 that live on port `p1`, which was declared back in Figure 5, and one client `lif` declared on lines 7–9 that is mapped to a hardware port `p2`. An `io_path` is constructed on lines 10 and 11 between `client_lif` and `server_lif1` using the NFS protocol to access the contents of `datastore nas_store` (declared back in Figure 6).

```
1 server_lif1 = lif(type="tcp_ip", port=p1,
2                   address="192.168.10.1",
3                   netmask="255.255.255.0")
4 server_lif2 = lif(type="tcp_ip", port=p1,
5                   address="192.168.11.1",
6                   netmask="255.255.255.0")
7 client_lif = lif(type="tcp_ip", port=p2,
8                  address="192.168.10.2",
9                  netmask="255.255.255.0")
10 io_path(type="nfsv3", datastore="nas_store"
11         end_points=(server_lif1,client_lif))
```

Figure 7: NFSv3 Mount Expressed as SLSL `io_path`

3.3 Workload Modeling Phase

The third automation phase described back in Figure 3 is comprised of tasks aimed at modeling the *work* being generated and measured. We identify two main tasks in this phase that need to be configurable using SLSL elements: ***(i)*** choosing dataset access paths ***(ii)*** describing the work to perform on the system under test.

3.3.1 Choosing Dataset Access Paths

Previously in the *Test Configuration* phase, we used SLSL `io_path` elements to configure the paths that *can* be taken to access the dataset. The first task of the *Workload Modeling* phase is to decide which of the configured paths should be taken to access the dataset. Modern storage architectures contain multiple paths to the same data, with some paths being *active* or *optimal* and some being *passive* or *sub-optimal*. This "multipath" approach allows for higher availability of systems, *i.e.,* they exhibit tolerance to failures because requests can be re-routed via *passive* paths when *active* paths go off-line; or, the systems can be configured for higher performance, meaning all paths can be configured as *active* and requests can be sent to the system according to an optimal load-balancing algorithm. Table 4 shows how controlling dataset access patterns is useful in storage performance testing.

Property	Uses in Performance Testing
dataset access patterns	create edge-case scenarios test capabilities of a link/port create failure scenarios minimize run-to-run variability

Table 4: Effects of Access Patterns on Storage System Performance

To allow the test designer to choose a dataset access pattern, SLSL provides a `path_mapper` abstraction consisting of an algorithm and a list of configured `io_paths`. The `io_paths` are reordered according to the underlying algorithm so that the work will arrive at the system under test according to the desired access pattern. Some algorithms are created manually and designed to test a specific scenario, while other algorithms may utilize a third-party multipathing I/O solution provided on the storage clients [6, 10, 13]. Figure 8 shows an example of a `path_mapper` declaration that uses a `round_robin` algorithm to order the `io_path` declarations `mount1`, `mount2`, and `mount3`.

```
1 path_mapper(type="round_robin",
2             io_paths=[mount1, mount2, mount3])
```

Figure 8: SLSL Round-Robin `path_mapper`

3.3.2 Describing Workloads

The final responsibility of the *Workload Modeling* phase is creating the description of the work being used to evaluate the performance of the system under test. SLSL provides the test designer with the ability to specify properties of workloads, like their models (*e.g.,* open, closed, or hybrid [19]), access pattern, or operation mix. There are seemingly countless possible types of workloads that can be presented to a system. However, most common workloads can be categorized by one of the five descriptions below:

- **microbenchmarks**: basic load generators that exercise specific functionality (*e.g.,* SIO [15], Iometer [18], and Iozone [23])
- **application benchmarks**: vendor-neutral and industry standard benchmarks that simulate a particular application workload (*e.g.,* SPC [22], SFS [21], and JetStress [9])
- **workload languages**: provides flexibility in building custom microbenchmarks or application benchmarks using a more general workload modeling language (*e.g.,* FileBench [12] and fio [4])
- **applications**: non-simulated workload from a live application
- **system workload**: workload that is internal to the system and often interferes with user-perceived performance (*e.g.,* RAID parity reconstruction, virus scans, backup operations, etc. . .)

SLSL supports a variety of these workload descriptions using a `workload` abstraction. SLSL `workloads` use one or more `path_mapper` declarations to determine the access pattern. Figure 9 illustrates these concepts using two different workload descriptions: 1 `sio` microbenchmark, declared on lines 1–6, configured as a single-threaded sequential read process, and 1 `spc1` application benchmark, declared on lines 7–13, configured with a target I/O rate of 5000 operations per second.

```
1 workload(type="sio", num_processes=1,
2          warmup=480, runtime=600,
3          path_mapper=nas_paths,
4          options={"read_pct":100,
5                   "rand_pct":0,
6                   "threads":1})
7 workload(type="spc1",
8          target_iops=5000,
9          runtime=600, warmup=480,
10         path_mapper_asu1=map1,
11         path_mapper_asu2=map2,
12         path_mapper_asu3=map3,
13         javaparms="-Xmx512m -Xss256k")
```

Figure 9: SLSL `workload` Declarations

3.4 Measurement Phase

In the previous automation phases we configured the testbed hardware, created the datastores, defined the possible I/O paths to the dataset, and configured our workloads to model the work that is going to be measured on the system under test. The final phase of automation, shown back in Figure 3, involves running the performance test and collecting measurements. We identify two high-level tasks of the *Measurement Phase*: ***(i)*** configuring performance test runtime ***(ii)*** configuring measurement data.

3.4.1 Configuring Performance Test Runtime

A typical runtime scenario for a performance test consists of several *iterations* of "start workload and take measurement" cycles. Often attributes of the workload (*e.g.,* add or remove load, change access patterns, etc. . .) or properties of the system under test (*e.g.,* create interference, create a system failure, etc. . .) are changed between iterations so that the system behavior can be observed. SLSL provides flexibility in configuring the runtime of performance tests by allowing test designers to specify the following key properties:

- **number of test cycles**: how many *iterations* of "start workload and take measurements" cycles are to be executed?
- **workloads**: what workloads should be run in parallel during a particular test cycle?
- **custom user actions**: what controls are provided to allow users to execute *custom actions* that change system properties before, during, or after a test cycle?

SLSL provides the `iteration` and `test_runner` elements to allow the user to define a list of time-bound test cycles we call *measurement intervals*. Figure 10 shows a measurement interval for one `iteration` as the elapsed time between t_x and t_y, where t_x is a time-stamp when all workloads have started, and t_y is a time-stamp when the first workload has completed.

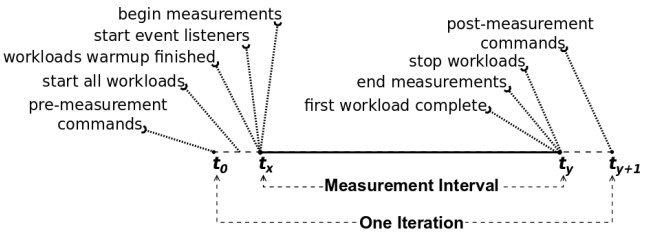

Figure 10: Timeline of one SLSL `iteration`

Iterations are comprised of a collection of workloads that will be executed in parallel and bound by a measurement interval. The `test_runner` element allows the user to specify the order in which multiple iterations will be executed. Lines 1–4 in Figure 11 show two SLSL `iteration` declarations, `itr1` and `itr2`, each containing 2 workloads. The `test_runner` declaration on lines 5–6 will result in the execution of iteration `itr1` first, followed by iteration `itr2`.

```
1 itr1 = iteration(workloads=[spc1_workload,
2                             sio_read])
3 itr2 = iteration(workloads=[spc1_workload,
4                             sio_write])
5 test_runner(name="OLTP with Interference",
6             iterations=[itr1, itr2])
```

Figure 11: SLSL `test_runner` Declaration

The behavior of the system under test can be altered during test execution using SLSL `command` and `event` elements. SLSL `command` elements provide a mechanism for specifying arbitrary commands to execute over command line interfaces. Command specifications are carried out from within pre-defined call-out points during an iteration. Two such call-out points are denoted on the time-line in Figure 10 at time t_0 as "pre-measurement commands", which are commands issued before the start of the measurement interval, and at time t_{y+1} as "post-measurement commands", which are commands issued after the measurement interval is over. SLSL `event` elements provide further customization by allowing the user to define python function *handlers* that will get invoked when the event occurs during the measurement interval. All event listeners are started at the beginning of the measurement interval as shown in Figure 10 at time t_x by the "start event listeners" call-out. Lines 1 and 2 in Figure 12 show a `command` declaration used to invoke an Data ONTAP® feature that will create a point-in-time *snapshot* on volume `vol1`. This snapshot will be created before the measurement interval of iteration `itr1`, because it is specified in the `pre_commands` list on line 8. The `event` on lines 3–6 uses a `timer` that will fire 30 seconds into the measurement interval of iteration `itr1`, at which point the handler defined by the `delete_snapshot()` function on lines 11-19 will get executed.

```
1  cmd = command(node=n1,
2                msg="snap create -V vol1 snap1")
3  evt = event(type="timer", delay="30",
4              handler="delete_snapshot",
5              kwargs={"snap_name":"snap1",
6                      "vol":"vol1"})
7  itr1 = iteration(workloads=[sio_read],
8                   pre_commands=[cmd],
9                   events=[evt])
10 # Define delete event handler
11 def delete_snapshot(**kwargs):
12     controller =  kwargs["controller"]
13     log = kwargs["logger"]
14     snap_name = kwargs["snap_name"]
15     vol = kwargs["vol"]
16     snap_delete = ("snap delete -V " +
17                    vol + snap_name)
18     log.info("deleting " + snap_name)
19     controller.send(snap_delete)
```

Figure 12: SLSL `command`, `event`, and Example Event Handler

3.4.2 Configuring Measurement Data

The second task in the *Measurement Phase* involves collecting measurement data necessary to determine whether or not the test meets the pass/fail criteria. The measurement data is consumed directly by analysts, or used as input to applications that aid in analysis by presenting a variety of views into the data. Since most analysis is done after test execution, it is important that the test be configured to collect relevant and accurate measurement data. Depending on the requirements of the analysis, there are many different types of measurement data ranging from high-level system reports (*e.g.,* Linux SYSSTAT Utilities [5]) to low-level system data (*e.g.,* function call-graph profiling [14], traces, raw OS counters, and core dumps). We must take into consideration that many of these measurement data collection mechanisms have an impact on system performance. Therefore, it is important to configure measurement data collection so it bears minimal impact on system performance and also contains all data needed to accomplish the analysis. SLSL provides a `result_collector` element that can be tailored to collect measurement data for a particular system, like Data ONTAP®, Linux, etc. . . , and it can be configured to include or exclude specific sets of data depending on analysis requirements. SLSL `event` elements can also be used to trigger measurement data collection based on events that occur during test execution. For example, data can be collected periodically using timer events, or conditionally using events that are triggered by specific conditions in the system under test.

3.5 Implementation

One of our main requirements of a little language implementation is that it be *lightweight* with respect to maintenance and extensibility. We don't want SLSL to be bogged down with complexities introduced by compiler technologies, which is a trait typical of other little language approaches [1]. We also don't want to focus our efforts on implementing general-purpose language control structures (*e.g.,* loops and conditional statements), but we want to make use of these control structures in SLSL when they are needed. For these reasons we chose to leverage existing python language components to implement SLSL. The SLSL parser is built as an extension of the python `parser` module [16], making SLSL a superset of the python language. SLSL users are granted access to all python language features *for free*, and developers working on SLSL extensions can focus their efforts solely on the implementation of the language elements. This choice of implementation also facilitates the application of SLSL semantics because the underlying core automation libraries are also written in python. The SLSL translation layer relies on the python `inspect` module [17] to programmatically translate SLSL specifications into automation tasks using introspection [11].

4. CASE STUDY

We demonstrate the efficacy of SLSL within the context of multi-tenant, storage cloud applications. We base this case study on these applications because they provide great working examples that showcase the richness in possible test configurations afforded by SLSL. Storage multi-tenancy relies on a variety of client and server virtualization technologies [3] that are applied at a *logical component* level and can lead to complex test scenarios due to the sheer number of possible variations in system configurations. For example, a common storage multi-tenancy configuration would include many tenants with each "tenant" residing within its own storage vserver and having its own independent set of requirements on the physical hardware, configuration of datasets, data access patterns, storage policies, and application workloads.

For the purpose of this case study, our multi-tenant storage application example consists of two "tenants": ***(i)*** the first "tenant", whom we'll refer to as *NAS tenant*, represents a group of users accessing home directories served over NFS ***(ii)*** the second "tenant", whom we'll refer to as *SAN tenant*, represents an online transaction processing (*OLTP*) database application being served from a SAN over FCP. The objective of our performance test is to observe the performance of the system when each storage tenant is active in isolation and when both tenants are active in unison, while gradually increasing the system load as the test progresses. The rows in Table 5 list the requirements this objective places on the respective automation phases shown back in Figure 3.

To meet these requirements we use a combination of the SLSL constructs shown in Table 1 and python language elements. Figure 13 is a complete implementation in 54 lines of code of a performance test that meets all of the requirements outlined in Table 5. For the sake of completeness, lines 2–12 contain example declarations needed to describe a testbed with hardware components consisting of one Linux client (`c1`) with an Ethernet port (`eth2`) and two FCP initiators (`2a` and `2b`), two storage controllers (`n1` and `n2`) with one Ethernet port (`e0b`) and one FCP port (`4a`). Finally we construct a Data ONTAP®cluster (`my_cluster`) using the 2 controller nodes (`n1` and `n2`) on line 12.

Phase	Requirements
Test Configuration	2 volume containers 1 virtual server for each tenant 1 SAN datastore 1 NAS datastore
Workload Modeling	an OLTP benchmark using SAN a file-system benchmark using NAS
Measurement	measurement with *NAS tenant* active measurement with *SAN tenant* active measurement with both tenants active vary system load across measurements

Table 5: Multi-tenant Storage Test Requirements

```
1  # Define Hardware Configuration
2  c1 = client(type="linux")
3  c1_eth2 = port(name="eth2", type="ethernet", node=c1, speed="10gig")
4  c1_2a = port(name="2a", type="initiator", node=c1, speed="4gig")
5  c1_2b = port(name="2b", type="initiator", node=c1, speed="4gig")
6  n1 = controller(type="ontap8", model="FAS6080")
7  n1_e0b = port(name="e0b", type="ethernet", node=n1, speed="10gig")
8  n2 = controller(type="ontap8", model="FAS6080")
9  n2_4a = port(name="4a", type="target", node="n2", speed="4gig")
10 a1 = aggregate(node=n1, name="aggr1", num_disks="84", raidsize="16")
11 a2 = aggregate(node=n2, name="aggr2", num_disks="84", raidsize="16")
12 my_cluster = cluster(nodes=[n1, n2])
13
14 # Define Test Configuration
15 nas_vs = vserver(cluster=my_cluster, name="nas_server")
16 san_vs = vserver(cluster=my_cluster, name="san_server")
17 vol1 = volume(name="vol1", aggregate=a1, size="2t", server=nas_vs)
18 vol2 = volume(name="vol2", aggregate=a2, size="2t", server=san_vs)
19 nas_store = datastore(type="volume", container=vol1)
20 san_store = datastore(type="LUN", container=vol2, num_luns=100,
21                       lun_size="20g")
22 c1_nas_lif = lif(type="tcp_ip", port=c1_eth2, addr="192.168.10.1")
23 n1_nas_lif = lif(type="tcp_ip", port=n1_e0b, server=nas_vs,
24                  addr="192.168.10.2")
25 nas_path = io_path(type="nfsv3", datastore=nas_store,
26                    end_points=(c1_nas_lif, n1_nas_lif))
27 c1_san_lif1 = lif(type="fcp", port=c1_2a)
28 c1_san_lif2 = lif(type="fcp", port=c1_2b)
29 n2_san_lif = lif(type="fcp", port=n2_4a, server=san_vs)
30 san_path1 = io_path(type="fcp", datastore=san_store,
31                     end_points=(c1_san_lif1, n2_san_lif))
32 san_path2 = io_path(type="fcp", datastore=san_store,
33                     end_points=(c1_san_lif2, n2_san_lif))
34 nas_mapper = path_mapper(type="round_robin", io_paths=[nas_path])
35 san_mapper = path_mapper(type="multipath_linux",
36                          io_paths=[san_path1, san_path2])
37
38 # Define Workloads and Test Execution
39 test_cycles = 3
40 start_tenant_ops = 5000
41 scale_tenant_ops = 5000
42 iteration_list = []
43 cur_op_rate = start_tenant_ops
44 for i in range(test_cycles):
45     sfs = workload(type="sfs", target_iops=cur_op_rate, warmup=300,
46                    runtime=900, mount_path=nas_mapper)
47     spc = workload(type="spc1", target_iops=cur_op_rate,
48                    asu1=san_maps, asu2=san_maps, asu3=san_maps,
49                    warmup=300, runtime=900)
50     iteration_list.append(iteration(workloads=[sfs]))
51     iteration_list.append(iteration(workloads=[spc]))
52     iteration_list.append(iteration(workloads=[sfs, spc]))
53     cur_op_rate += scale_tenant_ops
54 test_runner(name="multi_tenancy", iterations=iteration_list)
```

Figure 13: Multi-tenant Storage Performance Test

On lines 15–36 in Figure 13 the *Test Configuration* is specified, starting with vserver declarations on lines 15 and 16 for each "tenant". The vservers are later used to create volumes, declared on lines 17 and 18, that contain the NAS and SAN datastores declared on lines 19–21 and logical interfaces on lines 23 and 29 for *accessing* the two datastores.

Finally on lines 39–54 we define the necessary components to meet the requirements of the *Workload Modeling* and *Measurement* phases. For the *NAS tenant*, we use a `workload` declaration of type `sfs` on line 45 to utilize the SPEC SFS Benchmark — an industry standard network file-system performance benchmark [21]. Likewise on line 47, we use a `workload` declaration of type `spc1` to make use

of the SPC-1 Benchmark — an industry standard *OLTP* benchmark [22]. Notice on line 39 we introduce a python variable `test_cycles` to represent the total number of "cycles" to execute. In the `for` loop in lines 44–53 we define a test "cycle" as three iterations (lines 50, 51, and 53) of measurements where we activate the *NAS tenant* work only, *SAN tenant* work only, and both, respectively. The three iterations are appended to a Python list `iteration_list` which is attached to the `test_runner` declaration on line 54 to complete the specification.

The performance test implementation shown in Figure 13 exemplifies how a relatively few number SLSL elements can be used to implement a performance test that meets fairly complex testing requirements. From our experience, because of the expressivity captured within the language constructs and the flexibility in the possible arrangements of declaration references, our approach allows test developers to more rapidly and more reliably implement new performance tests than traditional methods that use general purpose programming languages to implement automated performance tests.

5. CONCLUSION

Our goal has been to facilitate the rapid implementation of complex, automated performance tests for data storage systems using language abstractions that hide the complexities of common performance test automation tasks. To that end, we presented our little language, SLSL, and highlighted its key language elements within the context of the specific automation tasks they were designed to facilitate. We supplied several small code examples throughout the paper to show how automation tasks can be fulfilled with SLSL. We also discussed how the implementation of SLSL meets our lightweight design goals with respect to its maintenance considerations and extensibility. Finally, we provided one complete SLSL implementation that was demonstrative of how SLSL can be used to achieve a high degree of flexibility in performance test configurations.

Future extensions of SLSL will focus on providing the test designer with more control over various aspects of a test's runtime. For example, current measurement intervals are based solely on time. We are discussing future extensions that will allow a test designer to specify measurement intervals based on *phases*. Phase changes can be triggered by an arbitrary set of conditions in the system (e.g., a desired CPU utilization is reached, a failure event has occurred, etc. . .).

We also plan to explore adding *gate* elements in support of more `event` types. Gates will be used as background tasks that check various system conditions over periods of time and signal some other component when that condition has been satisfied. We are planning to use gates to perform rudimentary automated analysis during test execution. Among other things, this analysis can be used to provide a "score" for the "goodness" of the measurements collected by a particular test. These "goodness checks" will be used to omit bad measurement data in the event of a undesired system behavior during test execution (e.g., incorrect port speed, undesired bottlenecks, etc. . .). This will aid greatly in performance regression analysis, as there are hundreds of automated test results collected each week.

6. ACKNOWLEDGMENTS

The authors would like to thank the rest of the development team, including George Dowding, Will Spearman, Jay Goldfinch, Steven Yap, and Gordon Young for their hard work and ingenuity. Additional thanks goes out to Pete Wyckoff and Gary Little for lending ideas and providing guidance in times when guidance was needed, and to Jill L. Ferguson for graciously assisting with the editing process.

7. REFERENCES

[1] A. V. Deursen and P. Klint. Little Languages: Little Maintenace? *Journal of Software Maintenace: Research and Practice*, 10(2), 1998.

[2] M. Eisler, P. Corbett, M. Kazar, D. Nydick, and C. Wagner. Data ONTAP GX: A Scalable Storage Cluster. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 23–23. USENIX Association, 2007.

[3] P. Feresten. Storage Multi-Tenancy for Cloud Computing. Website, 2010. `http://www.snia.org/cloud/CloudStorageMultiTenancy.pdf`.

[4] Geeknet Inc. fio. Website, 2010. `http://freshmeat.net/projects/fio/`.

[5] S. Godard. Welcome to the SYSSTAT Utilities Home Page. Website, 2010. `http://sebastien.godard.pagesperso-orange.fr/`.

[6] E. Goggin, A. Kergon, C. Varoqui, and D. Olien. Linux Multipathing. *Linux Symposium*, pages 147–168, 2005.

[7] J. W. Lloyd. Practical Advantages of Declarative Programming. In *Joint Conference on Declarative Programming*. GULP-PRODE'94, 1994.

[8] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[9] Microsoft. Using Jetstress to Test Disk Performance. Website, 2005. `http://technet.microsoft.com/en-us/library/aa998462(EXCHG.65).aspx`.

[10] Microsoft. Multipath I/O overview. Website, 2008. `http://technet.microsoft.com/en-us/library/cc725907.aspx`.

[11] P. O'Brien. Guide to Python Introspection. Website, 2002. `http://www.ibm.com/developerworks/library/l-pyint.html`.

[12] Open Solaris. FileBench. Website, 2010. `http://www.solarisinternals.com/wiki/index.php/FileBench`.

[13] Oracle and Sun Microsystems. Solaris SAN Configuration and Multipathing Guide. Website, 2010. `http://docs.sun.com/app/docs/doc/820-1931`.

[14] J. Osier. GNU gprof. Website, 1993. `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`.

[15] T. Powell. March Tool of the Month: Simulated IO (SIO). Website, 2006. `http://partners.netapp.com/go/techontap/tot-march2006/0306tot\_monthlytoolSIO.html`.

[16] Python Software Foundation. Website, 2010. `http://docs.python.org/library/parser.html`.

[17] Python Software Foundation. Website, 2010. http://docs.python.org/library/inspect.html.

[18] C. Querol, D. B. Dov, D. Scheibli, J. Eiler, M. Zhang, R. Riggs, R. Altherr, T. Harmon, and V. Degoricija. Iometer. Website. http://www.iometer.org.

[19] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 18–18, Berkeley, 2006. USENIX Association.

[20] D. Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[21] Standard Performance Evaluation Corporation. Website, 2010. http://www.spec.org/.

[22] Storage Performance Council. Website, 2010. http://www.storageperformance.org/home/.

[23] webmaster@iozone.org. IOzone Filesystem Benchmark. Website, 2006. http://www.iozone.org.