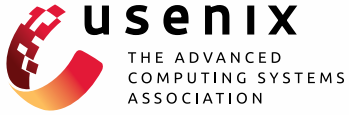


ParaSwift: File I/O Trace Modeling for the Future

Rukma Talwadker and Kaladhar Voruganti, *NetApp Inc.*

https://www.usenix.org/conference/lisa14/conference-program/presentation/talwadker

This paper is included in the Proceedings of the 28th Large Installation System Administration Conference (LISA14).

November 9–14, 2014 • Seattle, WA

ISBN 978-1-931971-17-1

Open access to the Proceedings of the 28th Large Installation System Administration Conference (LISA14) is sponsored by USENIX

ParaSwift: File I/O trace modeling for the future

Rukma Talwadker
NetApp Inc.

Kaladhar Voruganti
NetApp Inc.

Abstract

Historically, traces have been used by system designers for designing and testing their systems. However, traces are becoming very large and difficult to store and manage. Thus, the area of creating models based on traces is gaining traction. Prior art in trace modeling has primarily dealt with modeling block traces, and file/NAS traces collected from virtualized clients which are essentially block I/O's to the storage server. No prior art exists in modeling file traces. Modeling file traces is difficult because of the presence of meta-data operations and the statefulness NFS operation semantics.

In this paper we present an algorithm and a unified framework that models and replays NFS as well SAN workloads. Typically, trace modeling is a resource intensive process where multiple passes are made over the entire trace. In this paper, in addition to being able to model the intricacies of the NFS protocol, we provide an algorithm that is efficient with respect to its resource consumption needs by using a Bloom Filter based sampling technique. We have verified our trace modeling algorithm on real customer traces and show that our modeling error is quite low.

1 Introduction

Historically, benchmarks and traces have been used by system designers for designing and testing their systems. Designing new benchmarks corresponding to new emerging workloads, and getting them approved via standards bodies is a tedious and time consuming process. Thus, system designers usually use workload traces (each scoped to a particular system and application) to validate and verify their designs. However, in order to get a decent representation of an application, one typically needs to capture a trace for a few hours, and for many new emerging applications, this usually leads to the storing of multiple terabytes of data. There have been numerous proposals [6, 13–15] where traces have been leveraged to create workload models that can, in turn, be used to generate I/O patterns. The benefits of modeling traces are: 1) trace models are easier to copy and handle than large trace files 2) using the trace model one can scale the I/O patterns present in the original trace in both temporal and spatial domains and 3) it is much easier to simultaneously and selectively replay respective portions of the trace across respective server mount points using a trace model. It is important to note that a trace model provides a probabilistic representation of the original trace and is not an exact representation of the original trace.

Prior art in trace modeling has primarily focused on modeling block workloads (SAN workloads) [5, 7, 8, 11, 13, 15] and only one attempt had been made at modeling file (NAS) workloads [14] in the context of virtualized environments. In the virtualized environment file I/O workloads from the clients are served by the v-DISK layer of the VM and the underlying NAS storage device only receives the block based read/write requests. The resulting workload does not present many challenges inherent to the file I/O traces like the metadata and the I/O operation mixes, hierarchical file namespaces etc. Hence, modeling these virtualized file workloads is same as modeling SAN workloads. In this paper, we present algorithms and a framework for modeling and replaying NFS workloads that addresses many of the open problems that have been listed in a) the previous NAS trace modeling [14] and replay research efforts [18] b) the limitations of current file system benchmarking tools and c) also some new challenges in trace modeling due to the increased intensity in new types of workloads.

1.1 ParaSwift Innovations

ParaSwift makes the following contributions:

Representing Metadata operations and File System Hierarchies: Unlike in SAN trace modeling, in NAS trace modeling one has to accurately represent file metadata operations. For example, around 72% of operations in SPECsfs2008 benchmark are metadata related operations. Preserving the order of I/O and metadata operations matters to the application. Operations like creation and deletion of symbolic links, renaming of files/directories can change the underlying file system (FS) image. Accessing file handles via hierarchical

lookups on the file system image has not been handled in previous file system benchmarks. As a result benchmarks either work on a flat namespace or maintain a file to file handle path mapping in the memory. NFS file system benchmarking results are also quite sensitive to the structure of the FS image, the depth and breadth of FS tree, and the sizes of the various files and directories.

Workload Scaling: Workload scaling requires understanding and representing logical file access patterns so that workload scale-up operations can increase the load on specific files which might not be physically co-located on the storage media. Thus, there is a need to be able to observe the load intensity characteristics of the workload, and model it accordingly. Similarly, one needs to take the spatial and temporal scaling parameters as input, and subsequently replay using the trace model to satisfy the scaling requirements. Most of the file system benchmarks do not allow for this level of control with respect to scaling operations.

Handling New Workloads : Increasingly, new types of transaction intensive workloads are emerging where millions of transactions get executed per second (e.g. mobile ad applications, stock trading, real-time supply chain management etc). Storing file system traces at this scale would result in extremely large trace files. There are challenges with the amount of memory and compute resources required to 1) capture the stateful nature of the client-side protocol and 2) to learn the intricacies of a large number of file handles encountered in the trace and their off block boundary access size granularities. Trace model building algorithms that have been articulated in prior art [5,7,11,13] were not designed for handling these challenges. For instance, there is a minimum of 50x increase in workload intensity and number of clients per trace from the Animation workload traces captured in 2007 [12] vs. the one captured by us at our customer site in 2013. Hence, it is desirable to be able to dynamically create models as traces are being captured to reduce the resource requirements of tracing frameworks.

Lack of Similarity in Underlying File Systems: Usually, users of traces assume that traces collected in one setup (file system) can be used on a different file system. For example, the file mount parameters, read/write transfer sizes, and age of the file system can have substantial impact on the load being seen for the file server. Thus, it is important to leverage file traces in conjunction with the characteristics of the underlying file system. Apart from overheads of storing traces for on-demand replays, the existing trace replay tool [18] suffers from two other shortcomings 1) Replay timing accuracies suffer as the number of clients and NFS mount points in a single intense trace runs to hundreds (E.g. 640 client/server IP combinations in Animation trace of Table 1 on Page 9) 2) even under the same client workload, it is possible that different file servers based on the same protocol produce very different traces. Hence, replay of traces captured on one setup and replayed on another without inspecting the responses would not be useful.

In this paper we present a framework and algorithm called *ParaSwift* that can model file system traces (NFS), and it addresses the concerns that have been listed above. At this moment ParaSwift can be used to model and replay both NFS v3 and v2 protocols. The modeling code can be also used to model SAN traces but in this paper we are specifically focusing on NFS trace modeling.

2 System Overview and Algorithms

Figure 1 (Page 4) illustrates ParaSwift's model building architecture. One of the key design principles behind our thinking is that we need to be able to build models for very large traces in a resource efficient manner. This, in turn, forced us to re-think the steps involved in trace model building pipeline. ParaSwift's model building process is divided into 5 distinct phases:

Phase 1: Trace parsing : trace parser extracts and presents each NFS request and its corresponding response in the order of the request arrival to the trace sampling component.

Phase 2: Inline trace sampling : as a NFS request-response pair is streamed through ParaSwift, corresponding model building data structures have to be updated in-memory. Over a period of time the host memory will get exhausted and the respective data structures will spill over to the disk. This, in turn, will throttle the inline stream of NFS request-response processing. The inline trace sampling component helps in reducing the probability of the data structure spill over by discarding the redundant I/O's.

Phase 3: Inline trace model building : as the request-response pair is streamed through ParaSwift, in-memory data structures representing the various aspects of the trace are updated. The data structure allocation process has been designed to 1) make most efficient use of the host memory 2) eliminate the need for doing multiple passes through the trace for model building 3) represent the workload characteristics in a very succinct way without much information loss.

Phase 4: Batch trace processing : the in-memory data structures constructed by ParaSwift cannot be directly fed to a load regenerator. Few optimizations need to be done on the data structures for extracting the necessary model parameters. These procedures are essentially batch in nature.

Phase 5: Trace model translation for replay : For the purpose of workload replay we use a commercial load generator called *Load DynamiX*^{TM} [2]. *Load DynamiX* allows us to achieve a high fidelity translation of the

ParaSwift trace model to an equivalent workload profile. *Load DynamiX* also provides us with the ability to seamlessly scale the trace workloads to over a million clients per NFS server mount point.

In order to understand Phase 2 of the pipeline, one needs to understand Phases 3 and 4. Hence, below, we discuss Phase 2 after we discuss the other phases of the model building pipeline. We discuss each of the phases in the modeling pipeline in the following sections respectively.

2.1 NFS Trace Input and Parsing

NFS Trace : NFS protocol consists of two categories of operations. I/O operations that directly access or manipulate the contents of a file (NFS reads and writes) and metadata operations which read or write the file/directory metadata like getting file/directory access permissions (getattr), making file links (link, create link), setting directory/file ownership info (setattr) and updating filesystem information (fsinfo/fsstat). NFS traces are usually captured using OS tools like TCPDUMP in Linux and stored with a .PCAP or a .TRC extension. Since NFS traces tend to be bulky, efforts have been made to store them more efficiently in a compressed format known as the DataSeries, introduced in [4] and recommended by the Storage Networking Industry Association (SNIA). NFS trace stored in a DataSeries format (.ds) is essentially a database with all the information about the various operations (I/O as well as metadata) along with the corresponding network IP/port as well as the respective RPC packet timing information.

Trace Capture Tool : We have developed a tool within our group to capture NFS traces over network by mirroring network switch ports. This tool can handle rates of up to 16 GBps. For storing the NFS packets, the trace capture tool leverages the DataSeries software. Each incoming NFS packet representing a request or a response to a particular NFS operation is parameterized, compressed and stored as a separate entry in the corresponding operation specific DataSeries table for which the schema is predefined by the tool. The library stores the timestamp of each request/response in the order of their arrivals along with the corresponding network information in a separate DataSeries table. Trace replay is not part of this tool.

Trace Parser : ParaSwift cannot directly work on the compressed .ds traces. In order to extract a model, it has to un-compresses the respective DataSeries tables, pair the corresponding request/response operations, and reconstruct the temporal order of the requests. As we shall see later, pairing requests with the corresponding response helps in trace model correction. We have written approximately 350 LOC in DataSeries software distribution to accomplish the parsing and to generate an equivalent comma separated file (.csv).

Each line in the .csv file represents a request and its corresponding response along with the operation specific attribute values. The process of parsing a .ds file is called as Merge Join in ParaSwift as it's a join between the respective operation table and the table storing operation's timestamp and the network details in the DataSeries. Merge Join process is very light weight on memory as it operates on one row of each DataSeries table at a time and generates one request response pair at a time.

2.2 Inline Model Extraction

Each unsampled request/response pair extracted by the ParaSwift parser flows through all the components of the ParaSwift's inline processing pipeline as shown in Figure 1. We refer to these steps as inline, as they update in-memory data structures as the I/O's are streamed through the system.

2.2.1 Workload Stream and Pattern Separation

Due to host and storage virtualization, a NFS trace could contain requests coming in from multiple network clients going to multiple NFS mount points (identified by a host IP). Each distinct client-host IP combination is referred by ParaSwift as a stream, which is modeled separately and replayed simultaneously. This capability is not present in any of the benchmarks/replayers to date. Secondly, segregation of individual operation requests into macro level patterns per stream is essential from the user application perspective and hence the order of operations is more important than the proportions in which the various operations land on the NFS server.

Hence, this component of ParaSwift focusses on two aspects: 1) separate each request into a stream identifier based on its client and host network IP and 2) identify and separate individual operation requests per stream into macro level workflows called the workload patterns.

A workload pattern j belonging to a stream i ($WP_{i,j}$) is extracted based on: 1) maximum idle time between two consecutive request arrivals 2) maximum time elapsed between the previous response and the new request and 3) file handles involved in the two consecutive requests within a stream. For the two operations to be a part of the same workload pattern we do not mandate that they have to operate on the same file handle. For example, a physical client executing a Linux *make* task would involve multiple files. Each workload pattern extracted by ParaSwift is replayed separately during regeneration. The length and life-time of a workload pattern is

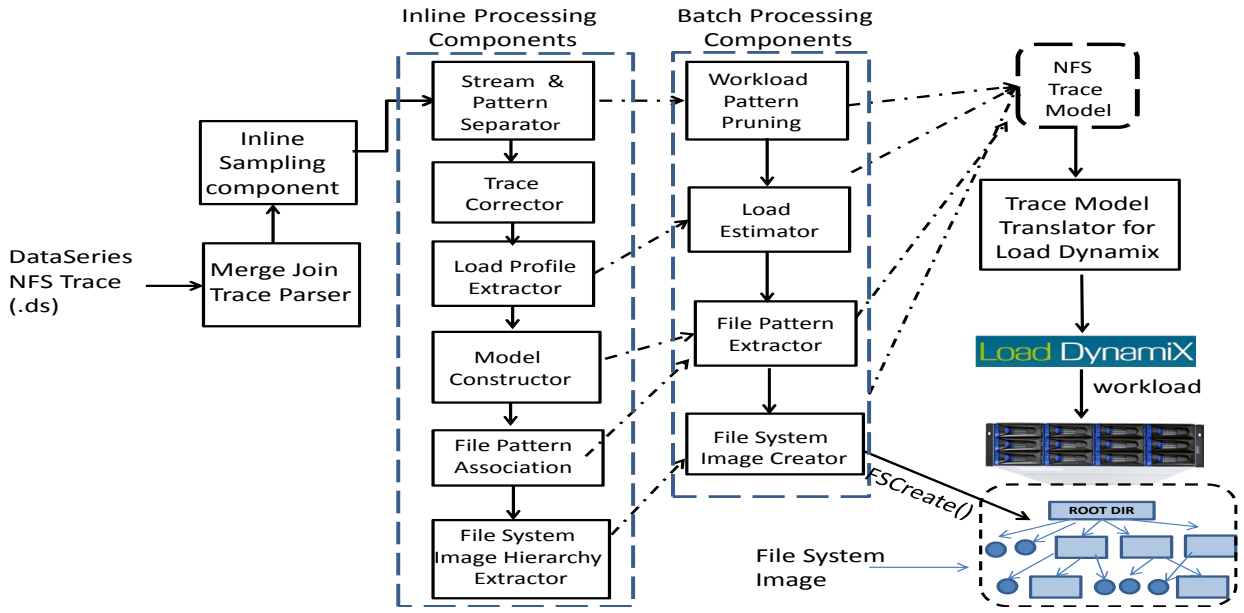


Figure 1: ParaSwift Architecture

recorded appropriately and exercised via specific knobs provided by the *Load DynamiX* replayer.

2.2.2 Trace Correction

One another aspect of trace modeling and replay for NFS workloads, is dealing with errors or I/O information losses. Whether the operation was missing or was mistakenly issued by the client can be inferred to a large extent from the corresponding NFS response. This is why a request and its response are presented together to the inline processing phase by the ParaSwift parser. In ParaSwift, we made a conscious decision to either correct the NFS request errors or to discard the erroneous operations without contaminating the final workload model.

If ParaSwift observes two successful consecutive remove requests on the same file handle, then in between the two remove requests there needs to be a create, rename, symbolic link, or link (hard link) request. If two operations cannot appear consecutively, we correct by additional operation insertion or remove the later request (if it was a failure). For operation insertion we too follow a table-driven heuristics approach similar to [18]. ParaSwift's trace correction scope is restricted within a workload pattern. Trace correction happens per request and with respect to the previously parsed request. A workload pattern is registered only post correction.

2.2.3 Load Profile Extraction

Each of the previously derived workload patterns need to be associated with load profiles. It means that streams which are more intense and perform operations at a faster rate than others need to be represented appropriately. Secondly, some workload patterns might be more bursty. The reasons for this behavior could be that some patterns might be script driven (e.g. Linux make task), whereas, some others might be driven by humans (e.g. reading a balance sheet). Based on these requirements, ParaSwift updates the following two parameters inline: 1) request arrival rates per stream (C) and 2) request arrival rates per workload pattern per stream ($r_{i,j}$).

Average request inter arrival time ($RIT_{i,j}$) in a workload pattern j of stream i is derived inline every 5 seconds interval. Based on this the total number of NFS operations that would be generated ($r_{i,j}$) per interval, by a given workload pattern, is obtained. ParaSwift also computes 80^{th} percentile value of the max number of NFS operations seen in the storage system from a given stream per interval, referred to as C. These computations are revisited and refined during the batch phase.

2.2.4 Model Constructor

In ParaSwift architecture we separate the process of workload pattern identification from its storage. Workload patterns have to be registered in a memory efficient data structure. We could have saved each pattern sepa-

rately as identifier separated operation entries. However, the workload patterns could be many in number and there could be many details associated with the pattern like: the file handles used and the attribute values of the operations they contain. Thus, a lot of memory can be needed to represent these details.

We chose to represent each association between a pair of operations via a shared adjacency matrix structure per workload stream. There are about 21 different operations (including I/O and metadata) that ParaSwift recognizes with a unique index based on the NFS v2/v3 standards. If an operation with index y follows an operation with index x in the workload pattern $WP_{i,j}$, the corresponding entry in the adjacency matrix is incremented by one. In order to allow us to reconstruct a workload pattern from a matrix we also build two additional data structures: a workload pattern start list and an end list per stream. These lists contain operations with which the workload patterns can begin or end respectively with the corresponding probabilities. All these data structures are updated inline and the normalization of probabilities is done in the batch phase. Figure 2 describes some of these data structures. Figure 2 also points to additional datastructures like the $fileHandleMap$ used to account for the unique file handles accessed, and how they are linked with each other.

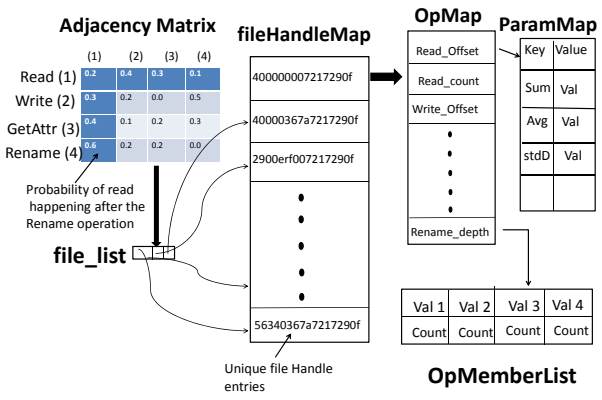


Figure 2: In Memory Data Structures in ParaSwift

2.2.5 File pattern association

The values of attributes for a particular NFS operation varies significantly even for a given file handle. For instance a file may not be always read from the same offset and for exactly the same number of bytes. Hence, it makes more sense to model them as distributions. The total number of unique files accessed and the type of operations performed on them is a huge set. Capturing these details in-memory was yet another challenge for ParaSwift.

Values of various attributes per NFS operation and the per file handle is recorded by ParaSwift in a memory resident data structure called the *OPMemberList*, which records each unique value of the attribute and the number of times such a value was reported in the trace separately. These values are later fitted to a distribution function per file handle and per operation, and written to the disk during the batch phase. In order to speed up processing, and reduce the number of compute passes over the data structures during the batch phase. A superset of parameters needed to compute the various distribution parameters like distribution averages are also partially computed inline and stored in a data structure named *ParamMap* as illustrated in Figure 2.

2.2.6 File System Image Exaction

ParaSwift reconstructs the essential File System (FS) hierarchy for the workload replay as-is from the trace. Generating FS image also happens inline and is stored in-memory. Since complete information about the exact location of the directory or a file in the FS may not be available till the end of parsing, we differ writing the actual FS image to the NFS server until the end.

We use a small in-memory representation called the *fstree* similar to the one used in [16] which stores just the right amount of information to create an FS image later. fstree contains pointers to file and directory objects. Each directory object only contains a pointer to its parent directory and its child directory/file objects and no other information. Each file object only contains information on its latest size in bytes obtained from the NFS operation response packet. During FS creation, the file size parameter is used by ParaSwift to create a file with equivalent number of bytes filled with random data pattern. We do not store directory/file i-node metadata such as access permission, ownership info etc. So far we do not use this information during replay.

2.3 Batch Model Processing

There are some aspects of trace model extraction which need to have a macro picture of the overall statistics in the trace. Such decisions cannot be made inline and have often necessitated multiple passes through the trace in the past [6, 11, 13]. We instead do multiple random accesses through few of the partially processed data structures built in-memory like the *OpMemberList* and *ParamMap* in Figure 2. These components are listed in Figure 1 and we discuss them in this section.

2.3.1 Workload Pattern Pruning

During the inline model extraction phase a large number of workload patterns are mined. This could be partly because these patterns are obtained heuristically. However, depending on the frequency of request arrivals in the workload pattern, not all of them might be significant enough to be regenerated.

ParaSwift defines a *Path_Probability_Threshold* which is the minimum expected probability of occurrence of a workload pattern for it to be included in the final model. The Path probability ($s_{i,j}$) of a given workload pattern is the cross product of start probability of the first operation (in the start list) in the workload pattern with the subsequent transition probabilities of other operations obtained from the adjacency matrix until an operation which also appears in the end list is encountered.

Algorithm 1 Workload Pattern Pruning Algorithm

```
for all Operation index entry in the start_list do
    Add the entry to IncompletePathsList sorted by
    the descending order of the start probability
end for
while exitcondition == false do
    BestPath = Get highest probability path from In-
    completePathsList
    x = index of the last operation in the BestPath
    for all y in AD_{i,j}[x][y] do
        if AD_{i,j}[x][y].count != 0 then
            BestPath.nextOp(y)
            BestPath.pathProbability *=
            AD_{i,j}[x][y].prob
            if end_list_i.contains(y) &
            BestPath.pathProbability >
            Path_Probability_Threshold then
                completePathsList.add(BestPath)
                totalProb = totalProb +
                BestPath.pathProbability
                M = M + 1
            else if ! end_list_i.contains(y) &
            BestPath.pathProbability >
            Path_Probability_Threshold then
                IncompletePathsList.addSorted(BestPath)
            end if
            if totalProb >= Coverage_Threshold ||
            M >= Total_Paths_Threshold then
                exitcondition = true
            end if
        end if
    end for
end while
```

Since adjacency matrix can also be realized as a directed graph, ParaSwift uses a breadth first search algorithm as listed in Algorithm 1 with greedy path pruning for extracting the most significant workload patterns. Every incomplete pattern is inserted into a sorted path list. The path with the highest probability is picked first and expanded. The algorithm terminates when either the total number of workload patterns extracted exceeds *Total_Paths_Threshold* (presently 25) or sum of the path probabilities for the paths extracted exceeds *Coverage_Threshold* (presently 70%). These parameters are tunable by the user based on the expected fidelity of the outcome.

2.3.2 Load Estimator

The two quantities, load intensity per workload pattern ($RIT_{i,j}$) and intensity per stream (C) calculated in the previous phase are based on averages, and hence, there is a possibility of under-loading the system. Secondly, workload pattern pruning step also eliminates some of the less frequent workload patterns which may further reduce the load intensity of the stream. During replay, in order to achieve a realistic load factor, ParaSwift utilizes two additional load related knobs provided by *Load DynamiX*. These include: concurrency factor per workload pattern ($p_{i,j}$) and total number of simultaneous clients per stream in the system (N).

With $s_{i,j}$ representing the path probability of a workload pattern $WP_{i,j}$ and M being the total number of significant workload patterns per stream i obtained from the previous step, values of $p_{i,j}$ and N are then computed as:

$\sum_{i=1}^{M} s_{i,j}$ ¡= 1 by the law of probability -(1)
$p_{i,j} = s_{i,j} \times N$ for i=1,..M - (2)
$N = \frac{C}{\sum_{i=1}^{M}(s_{i,j} \times r_{i,j})}$ - (3)

All of the load profile related parameters are represented at a granularity of 5 seconds over the entire period of observation. Any change in the values of these parameters across intervals is expressed as a corresponding ramp up/ramp down factor by ParaSwift during the Phase 5 of the pipeline illustrated in Figure 1.

2.3.3 File Pattern Extraction

In this step ParaSwift performs a function fitting per NFS operation attribute per file handle. To regenerate a value for an attribute (e.g. offset and I/O size for a read/write), instead of regenerating it empirically as in [14, 17] from the *OpMemberList* data structure ParaSwift does a least error function fitting of the parameters. ParaSwift is aware of a few standard distributions like the Constant, Gaussian, Normal/Uniform, Log normal, Exponential

and Polynomial. The least error function is chosen and the corresponding function parameters are extracted.

2.3.4 FS Image Creation and Hierarchical Namespace Lookups

ParaSwift reads the fstree created during the inline step and generates an appropriate FS image on the NFS server. Emulating a real client using benchmarks is a tough problem. NFS access begins with a lookup for the file handle in the correct FS directory path. This requires the client to be aware of its present path in the File System mount point. This could be accomplished by maintaining a file to file handle path map during replay. However, this technique does not scale as the working sets expand. To address this problem, ParaSwift creates every file seen in the trace at an appropriate location along with the corresponding soft link at the ROOT of the mount point during the fscreate() procedure. Load regenerator works on the links, but this ensures that every link read for a write operation translates into corresponding lookups and reads/writes for the actual file in the NFS server. However, few operations like file delete would still leave the actual file untouched during replay.

Scaling : ParaSwift provides an option of spatial scaling during the FS image creation. One can specify scaling of following two types: a) uniform scaling b) proportionate scaling. In uniform scaling ParaSwift simply adds more files in every logical path by a factor provided as an input. Associated soft links are also created accordingly. Proportionate scaling is the more interesting option provided by ParaSwift. ParaSwift at present supports few options like, "scale up metadata portion by x%". In this case ParaSwift already knows through OpMap (in Figure 2) scan processing which file handles are accessed for metadata versus which are accessed for I/O operations. Accordingly, it scales up the file count in the appropriate directories. For time scaling, respective load profile parameters are proportionately increased/decreased by ParaSwift

2.4 Trace Model Translation for Replay

Problem of workload regeneration is often delinked from its modeling. However, delinking the two aspects often results in gaps as discussed in section 1.1. We use *Load DynamiX* as a workload replayer in ParaSwift at the moment. *Load DynamiX* allows us to 1) emulate the probabilitic ordering of the various NFS operations within a workload pattern; 2) regenerate the various attributes of a NFS operation like NFS read size and offset etc. using Load DynamiX offered statistical functions; 3) scale infinitely; 4) exposes excellent knobs with respect to controlling request intensities; 5) does a time accurate request regeneration; 6) and supports i-SCSI (SAN), NFS and CIFS protocols. We use version 30.21444 of *Load DynamiX*.

Load DynamiX allows ParaSwift to specify per NFS operation specific attribute values as either averages or enumerations per file handle. Averages are not good enough and enumerations would not scale for larger traces. *Load DynamiX* additionally provides an ability to specify any distribution as a function of *Random* distribution function. Given a max and a min value, the Random function generates a random value within the range.

Based on the theory [10], each realization of a Random function would result in a different function. ParaSwift maps various distribution functions and their parameters to appropriate min/max values of the Random function to regenerate the respective distribution function.

Load DynamiX provides a Perl API module for creating a NFS workload project. The APIs support functionalities like 1) workload pattern creation; 2) respective load profile specification; 3) associating individual operations within a workload pattern to appropriate file handles, and operation parameters to an equivalent Random distribution parameters. We have written additional 700 LOC in *Perl* to leverage the *Load DynamiX* Perl module and generate an equivalent workload project by directly reading from ParaSwift workload model. Workload project can be directly imported via the *Load DynamiX* client GUI and replayed on the NFS server on which the FS image has been already created.

2.5 Inline Sampling

Our thinking behind building in-memory data structures is to speed up the trace processing times. We already see about 50x-100x increase in the request intensity between animation workloads captured in the year 2007 and those captured by our internal tool in 2013 listed in Table 1.

Workloads on the other hand may not change often. A trace data collected for first 5 minutes might be well representative of the later 10 minutes. If we could infer and exploit such self similarity in the traces *inline*, we can retain in-memory processing benefits at higher trace intensities. ParaSwift implements a novel inline content aware I/O sampling technique.

The technique is based on the concept of Bloom Filters (BF) [9]. BF is a space efficient, probabilistic data structure used to test containment of an object in the set. Efficiency of the implementation depends on the trace parameters used to test the containment. Our implementation is based on the following design constraints:

1. Intensity of I/O's needs to be retained : Sampling may make it appear that incoming I/O stream is very slow. ParaSwift model needs to preserve the original I/O

intensity.

2. Preservation of operation distributions : Filtering should not introduce a skew in the NFS operation distributions.

3. Compute and Memory Footprint : Data structures for containment have to have a low memory footprint and allow a quick lookup.

2.5.1 The Technique and Implementation

For the purpose of establishing the need for sampling, we merged several subtraces obtained from the IT workload trace capture to be discussed in section 3 as in Table 1. It took ParaSwift 4 hours to process a trace (.ds) worth 20 GB. We profiled this execution and learnt that the processing times could have be slashed by reducing the footprint of the model building data structures which were being extended to the disk as the number of unique files in the trace changed too often. We realized that it was not the number of unique files but the unique characteristics that matters for modeling. We sampled out the data at 60% (discarding 60%) with our technique and were able to complete the task in 1 hour with only 10% loss in fidelity.

For read and write operations, it is often important to look at the specific properties rather than just looking at the file handle. For instance, a random small read vs. a large sequential read, a file overwrite vs. partial write would have completely different performance implication. Hence, we look at the offset and the size characteristics of a read/write as the distinguishing factor. For metadata operations ParaSwift does not worry about the exact values of the attributes being set on the file or the directory i-nodes. In some cases like the rename or the create link operations the relative path or the location of the target with respect to the source file handle matters (from the file system performance point of view).

We adopt BF's only for detecting redundant read/write operations. BF lookup based on the containment function gives out a location index. If the corresponding bit index is already set, this implies that the value might already be there. BF can at times give false positives. However, there are no false negatives. Our design constraint is to minimize number of false positives (fp) for read/write operations.

We use BF fundamentals to decide: m, which is the number of bit locations a given function would hash to; k, which is the total number of hash functions we should use for a set of n input values. This is analogous to number of I/O's a trace would contain (each request-response pair being one I/O). We fix our fp rate to 10% which also lowers the bounds of the final model fidelity. We could reduce fp rate further, but we are convinced that its effectiveness depends on the variance of the parameters to be hashed in the trace, which are assumed to be uniformly distributed by the BF theory, and may not hold true for every workload trace.

ParaSwift uses a partitioned BF [9]. Our first hash function hashes offset and the second function hashes the size of a read. Similar BF's are designed for the write operation as well. Both of the containment functions work irrespective of request file handle. Semantically this implies that if a request makes access to a unique offset within a file and for a unique number of bytes, it is a new type of I/O irrespective of its file handle. This comparison indirectly favors I/O's which access files of non-identical sizes.

When sampling is turned on, the trace parser generates a random number per I/O to be processed. If the random number is within the expressed sampling rate, the request/response pair is presented to the sampling component. Whether an I/O is to be sampled out or not is decided by the BF for a read/write operation. For the metadata operation, only if it has not been recorded previously in adjacency matrix it is retained, else it is discarded. If it is a metadata operation to be discarded like rename, and create operations, ParaSwift records the relative path distance, as discussed earlier, that is applied during the replay. For all operations, the corresponding adjacency matrix is always updated irrespective of the verdict to avoid an operation distribution skew. Also, the number of unique file handles per interval are recorded, though the corresponding file handle maps as seen in Figure 2 are not updated if the I/O is to be sampled out. This preserves the working set sizes. During model translation, appropriate number of pseudo files are created to regenerate the original working set.

The two (read and write) partitioned BF's each with two partitions account for 722 KB of total memory space. Space needed for target and source path distance accounting, for the metadata operations, is negligible. Both the hash computations and filter lookups are constant time operations. This makes sampling strategy pretty amicable to inline trace model building. We use a popular and a reliable Murmur hash function [1] for all the Bloom filters.

ParaSwift, excluding the model to replayable project translation part, has been completely written in $c++$ with approximately 2500 LOC.

3 Evaluation

In this section we evaluate ParaSwift against 3 different NFS traces. We compare the accuracy of ParaSwift in recreating various workload patterns/characteristics observed in these workloads. We verify whether ParaSwift can: A) recreate various operations (I/O as well as metadata) seen in the original trace in right proportions; B)

Table 1: NFS traces used for Validation

Label	Capture duration	Number of unique streams	Total capture size
IT	One week	114	22 TB
Cust1	One weeks	63	17 TB
Animation	13 Hours	640	1.7GB

recreate read/writes with appropriate file offsets C) have right I/O size distributions; D) emulate right file working set sizes with appropriate file access popularity distribution E) mimic identical read and write payloads to the NFS mount point per stream and finally F) to retain all of these capabilities while sampling the input trace at variable frequencies. As the techniques used in this paper are statistical in nature our goal in this section is to be as close to the original workload statistically and not identical. We shall refer to the above itemized list of metrics while presenting the experiment results (in the same order).

3.1 The Workloads

Table 1 lists the NFS traces used to validate ParaSwift. Traces labeled as IT and Cust1 are collected by our internal trace capture tool for a week's duration. IT being our internal IT workloads and Cust1 representative of a customer running version control and scientific simulation like workloads concurrently on separate NFS mount points but as a part of the same trace. Each subtrace (a portion of the larger trace) is less than 4 GB and depending on the I/O intensity, covers a duration anywhere between 8 to 15 minutes. There are 900 such subtraces for IT and 495 for Cust1 trace covering a one week duration. We also regenerated a subtrace (1 out of 473) from the Animation workload traces (*Set 0*) captured in 2007, and published at the SNIA site [12], described in Table 1. Detailed information about their capture and analysis can be found at [3].

Each of these traces had multiple clients and NFS mount points. ParaSwift extracted each combination as a separate stream, modeled the various streams in the same run of the ParaSwift and regenerated them simultaneously via *Load DynamiX*. Each trace was run for the same duration as the original trace. Having the capability of being able to emulate multiple streams simultaneously is an important contribution of ParaSwift. We quantify the accuracy of regenerations on a per stream basis.

3.2 System Configuration

ParaSwift runs on a Linux based VM with 15 GB memory extensible up to 64 GB with 100 MBps network connecting a NFS storage server. Load regeneration was accomplished using 1 GBps full duplex physical port of *Load DynamiX* which could scale to over a million virtual ports (streams). Since ParaSwift does not focus on reproducing the identical performance profile (latency, file system fragmentation) we use a standard storage server configuration with 6 GB memory, 1 TB volume with 7500 rpm disks and 1 GBps network for all trace regenerations.

We validate ParaSwift by comparing the original workload trace with regenerated workload trace captured by *Load DynamiX* at the time of its execution.

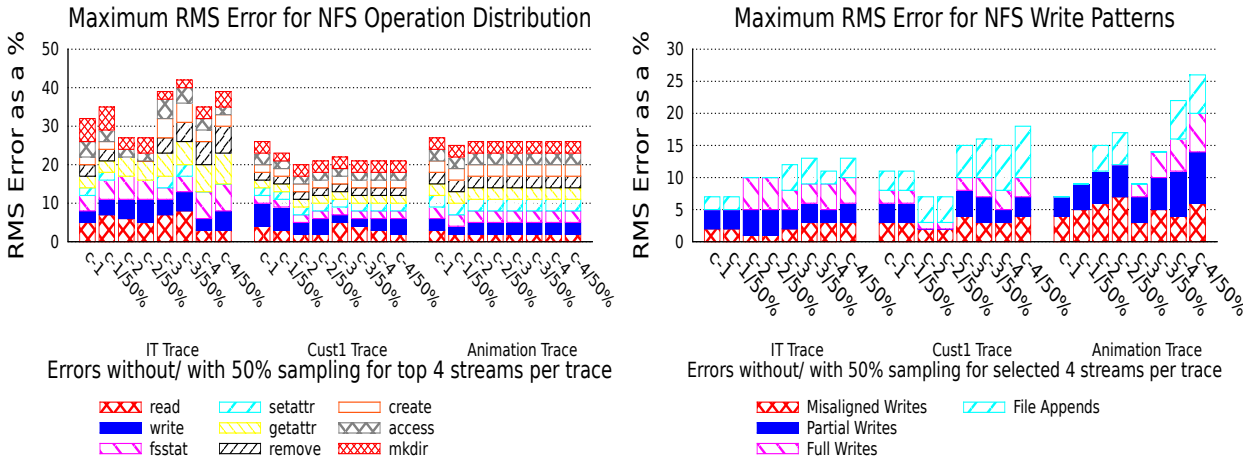
3.3 Experiments

In this section we present evaluation results based on three traces: IT, Cust1 and Animation as described in Table 1. We perform per metric statistical comparison of the original trace with the *Load DynamiX* collected synthetic trace for all of the earlier mentioned dimensions. We represent error as the Root Mean Square (RMS) error which is the standard deviation of the differences between regenerated values and original values of a metric in its respective distribution buckets.

3.3.1 Choosing subtraces

Since the total number of subtraces collected for IT and Cust1 workloads is huge, for the purpose of validation we sampled 20 subtraces belonging to different durations of the day and week. For Animation workloads we modeled one subtrace out of the 473 published subtraces. We compute metric errors per stream in a subtrace separately. In this paper we report results for the top 4 streams in a single subtrace for all three workloads. However we have validated that the various metric error bounds hold true for other streams in the three traces as well as for the other randomly chosen 19 subtraces for IT and Cust1 workloads. According to our preliminary analysis of 3 workloads in Table 1, workload patterns for these different traces varied quite a bit. Therefore, even though we only had 3 traces, we actually had many subtraces with vastly different characteristics.

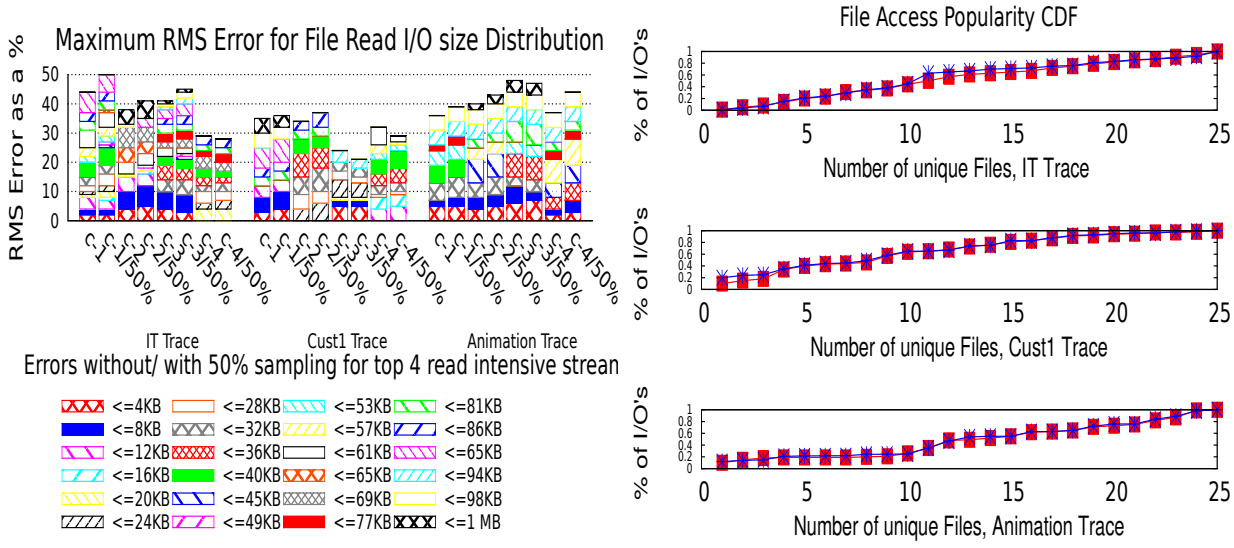
A) Figure 3(a) represents the RMS error for distributions of various operations for chosen streams in the each of the three traces. Streams which 1) had the highest request rate for chosen capture interval and 2) demonstrated wide variation in the type of NFS operations issued were chosen. For each stream (denoted as symbol C in the graphs) we also compared the corresponding errors when the original trace (.ds) was sampled at 50% (denoted as C-50%) of the actual trace size using ParaSwift's Bloom Filter (BF) technique. We see that max RMS error for all of the streams for all the metrics in Figure 3(a) is below 10%. Also there is not much dif-



(a) I/O and Metadata Operation Distribution Errors

(b) Write Pattern Distribution Errors

Figure 3: RMS Error for various distributions for the top 4 streams of each of the 3 workloads



(a) Read I/O size Distribution Errors

(b) File Access Popularity CDF

Figure 4: RMS Errors for Read I/O size Distributions and CDF of File Access Popularity

ference in the RMS errors between a sampled and an unsampled version of the trace model. Low errors in operation distribution are attributed to the Markov chain representation (adjacency matrix) and later the smart path pruning algorithm constituting ParaSwift's batch model processing phase. Also note that the the overall value of the y axis for each bar graph is a cumulative sum of the individual RMS values.

B) Figure 3(b) lists the RMS error for a distribution of various write patterns regenerated by top 4 streams at varying file offsets per a workload trace. Any read/write which does not happen at the 4 KB block boundary is misaligned for us. Any write which starts at offset 0 and ends at offset equal to file size prior to the operation (obtained from the response of the operation) is an overwrite. Any write which begins at the end offset of a file is an append. Any write not satisfying the latter two criteria is a partial write. For this comparison to be effective, streams which were intense as well as had a high fraction of writes were chosen. As seen in Figure 3(b) ParaSwift is able to fairly mimic the various write patterns restricting the RMS errors within 10%.

C) We also computed RMS errors for read/write offset as well as I/O size distributions (NFS attribute count). For comparisons we chose streams which operated with distinct I/O sizes. We associate each 4 KB offset boundary as a separate bucket for all the 4 metrics. We considered buckets up to 100 MB, but in Figure 4(a) due to space constraints in the graph we only illustrate read I/O size distribution RMS error for request sizes of up to 1 MB. The RMS errors were bounded by 8%. More importantly, the I/O sizes not occurring in the original trace were never regenerated by the *Load DynamiX* streams. The same holds true for rest of the metric results as well. Results were bounded by similar error rates for the other distribution buckets as well. The reason for choosing 4 KB aligned distribution boundaries is that irrespective of whether a read is for a full block or for a few bytes within a block, our NFS server always reads a full block.

ParaSwift's success in Figures 3(b) and 4(a) is due to least error distribution fitting during the model building phase and the efficient translation of the corresponding function into an appropriate *Random* function parameters for *Load DynamiX*. ParaSwift's sampling technique carefully designs the BF functions to retain any variability in I/O size and file access offsets while sampling. Competitive RMS error values of the respective parameters for each stream against the 50% sampling, establishes the efficiency of our BF design.

D) Figure 4(b), illustrates the CDF of the file access popularity per a chosen stream of each workload. RMS errors of the total number of unique files per every 5 seconds interval were limited to 5% for all the sampled trace streams. Figure 4(b) illustrates the CDF for the top 25

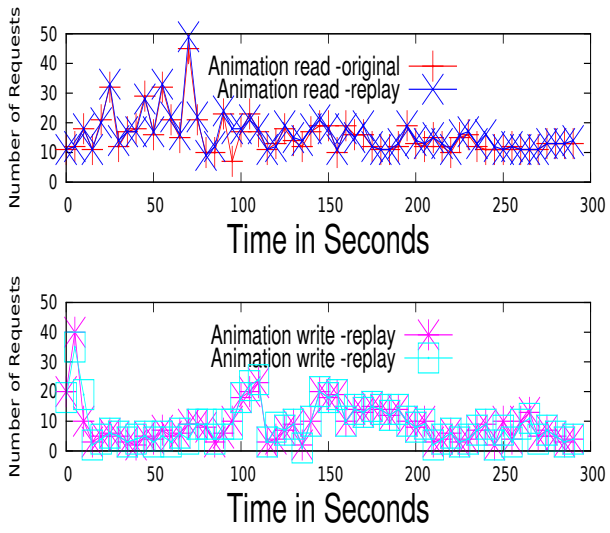


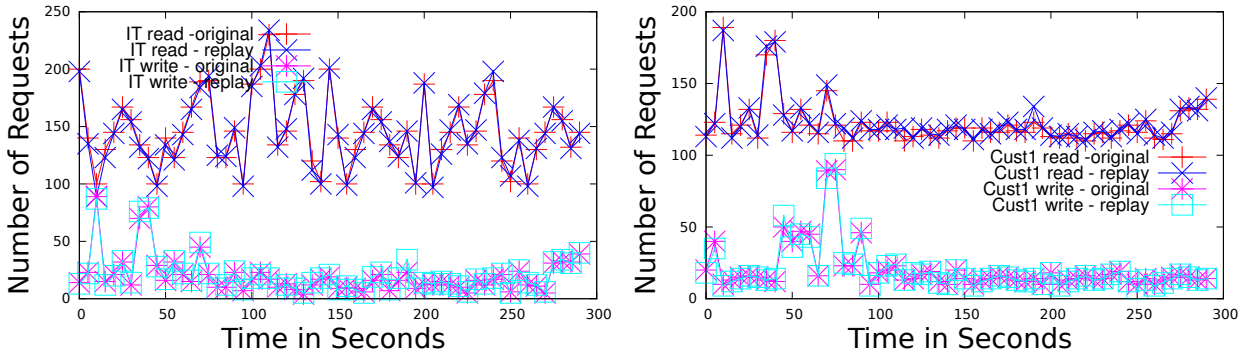
Figure 6: Animation Workload: Read and Write Requests, Original and Regenerated over a 5 minutes interval

files in the working set for a randomly chosen interval in the trace. We plotted the actual CDF vs. the replay CDF and found the errors to be less than 6%. Emulation of exact working set is accomplished due to recording of active file handles per operation of the adjacency matrix in the fileHandleMap data structure illustrated in Figure 2.

E) Figures 5(a), 5(b) and 6 represent comparison of the number of read/write requests generated for every 5 seconds interval by the most intense streams per workload trace. In reality, every new request issued by the *Load DynamiX* appliance is synchronous and response driven. However, the traces used by ParaSwift are from real deployments where the response is driven by the actual state of file system buffers, on disk fragmentation, and many other factors. In this experiment we want to assert the capability of our NFS client to generate the same intensity of read/writes when the various conditions pertaining to FS image and its state are identical. Hence, we turned off synchronous I/O mode and replay error notification in *Load DynamiX*.

We observe that the replay request rate closely follows the original rate for both read and write operations as the RMS errors were lower than 8%. This is attributed to ParaSwift's detailed calibration of load profiles in the form of 1) total number of workload patterns and 2) number of concurrent streams per workload pattern.

As mentioned earlier, these results hold true for other streams in the respective traces as well as for the larger sampled set.



(a) IT workload (b) Cust1 workload

Figure 5: Read and Write Requests, Original and Regenerated over a 5 minutes interval

3.3.2 Sampling and ParaSwift Performance

F) Figures 7(a), 7(b) and 8 show the error vs. processing times for the corresponding streams of each workload with varying sampling frequency. Due to space constraints, we represent the error in these cases as the maximum of the RMS errors for metrics in A, B, C, D discussed above.

These figures show a slight drop in the accuracy (less than 10%) for up to the sampling rates of 60% with at least 5X boost in the processing speeds for all of the workloads. Beyond 60% we found the errors to be unbounded. ParaSwift can operate in two sampling modes: conservative and progressive. In the conservative mode, the actual sample in size may be more than expected as the requests could not be sampled as they were found to be unique by the BF. In the progressive mode, if the number of outstanding requests to be sampled exceeds a threshold, we discard the request irrespective of its novelty till the outstanding bar is pushed below the threshold. The results above come from the progressive mode. However, based on the physical characteristics of the trace, progressive sampling may degrade to a random sampling over time.

The same file handle can be accessed by multiple workload patterns simultaneously. Replay errors most likely result from such situations. The number of such situations cannot be either theoretically estimated or proved experimentally. Hence, we try to avoid such errors by advising *Load DynamiX* to re-create a new file of the same name if it does not exist in the FS image (at the time of its access). This is an inline replay correction method that we embed during the translation of the ParaSwift workload model to a *Load DynamiX* replayable project.

Finally, all the experiments reported in this paper were repeated thrice and the average reading was considered. Every replay was done with all the concurrent streams

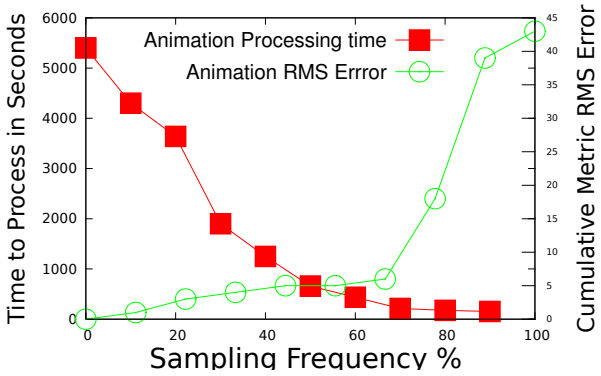
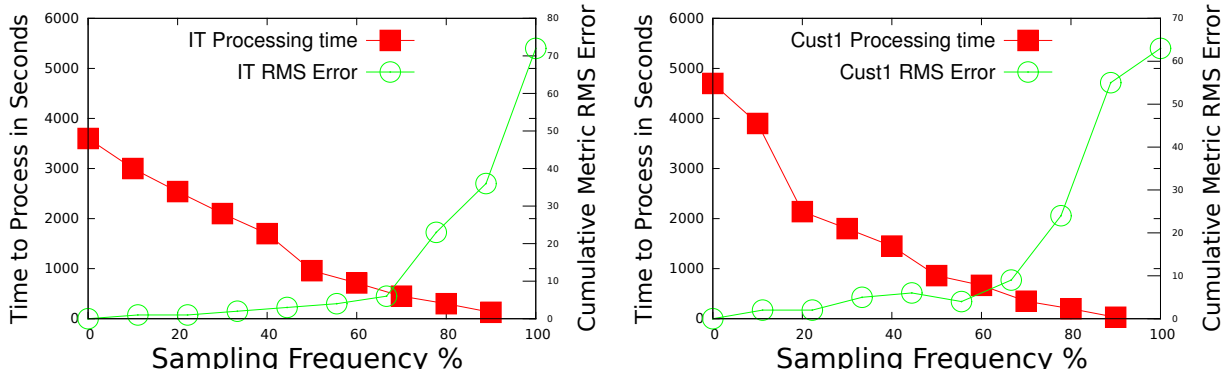


Figure 8: Animation Workload: RMS Error Vs. Compute times for the most intense streams with varying VM memory sizes

seen in the original trace run simultaneously as captured by ParaSwift load profile extractor in a dynamic way. Besides the RMS errors, the maximum errors in all of the above experiments were well below 12%.

4 Future Work

Statistically significant regeneration of multi-dimensional workload characteristics for NFS and SAN workloads is what we have primarily achieved via ParaSwift. However there are other aspects of workload regeneration essential for reproducing specific performance issue with trace replay. File system fragmentation is an important part and parcel of this problem. Another aspect deals with NFS file locking. While each of these problems are important by themselves they need a focussed investigation. ParaSwift is presently being used in our organization for reproducing customer problems dealing with workload characteristics alone, to validate various caching optimizations for various workload



(a) IT workload

(b) Cust1 workload

Figure 7: RMS Error Vs. Compute times for the most intense streams with varying VM memory sizes

verticals and stress testing our systems with specific workload segments. Appropriate aging of the file system and advanced features associated with stateful versions of NFS (V4) are a part of our future investigations.

5 Conclusion

In this paper we presented an algorithm/tool called ParaSwift that creates workload models based on traces, and then helps to replay them. This work addresses many of the open problems in the trace modeling and replay arena. ParaSwift implements a novel Bloom Filter based inline trace sampling mechanism that helps to both reduce the model creation time and also reduce the amount of memory that is consumed during the model building process. Our intention is to open source this code and enable the research community to build and distribute a large library of trace models that can be used to test future systems. Look out for the package ParaSwift on the world wide web in the near future. Finally, we have tested our end to end workflow using many real life traces and our results show that our techniques in general lead to less than 10% error in the accuracy of the trace model.

References

[1] https://sites.google.com/site/murmurhash/.

[2] Load dynamix, inc. http://www.loaddynamix.com/.

[3] E. Anderson. Capture, conversion, and analysis of an intense nfs workload. In *Proccedings of the 7th Conference on File and Storage Technologies, FAST '09*.

[4] E. Anderson, M. Arlitt, C. Morrey, and A. Veitch. Dataseries: an efficient, flexible, data format for structured serial data.

[5] Peter B, Armando F, Michael J. F, Michael I. J, and D Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*.

[6] D. Christina, S. Sriram, K. Badriddine, V. Kushagra, and K. Christos. Time and cost-efficient modeling and generation of large-scale tpcc/tpce/tpch workloads. In *Proceedings of the Third TPC Technology conference on Topics in Performance Evaluation, Measurement and Characterization*.

[7] G. Ganger. Generating representative synthetic workloads: An unsolved problem. In *Proceedings of the Computer Measurement Group (CMG) Conference, 1995*.

[8] Maria E. Gomez and Vicente Santonja. Self-similarity in i/o workload: Analysis and modeling. In *Proceedings of the Workload Characterization: Methodology and Case Studies*.

[9] F. Hao, M. Kodialam, and T. V. Lakshman. Building high accuracy bloom filters using partitioned hashing. *SIGMETRICS Performance Evaluation Review, 2007*, 35 (1).

[10] M. Hazewinkel. Random function. Encyclopedia of mathematics, springer. 2001.

[11] Z. Kurmas, K. Keeton, and K. Mackenzie. Synthesizing representative i/o workloads using iterative distillation. In *11th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2003)*.

[12] SNIA. Iotta trace repository. `http://iotta.snia.org/tracetypes/3/`.

[13] R. Talwadker and K. Voruganti. Paragone: What's next in block i/o trace modeling. In *Proceedings of the Mass Storage Systems and Technologies, MSST '13*.

[14] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok. Virtual machine workloads: The case for new benchmarks for nas. In *Proceedings of 11th USENIX conference on File and storage technologies, 2013*.

[15] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[16] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the USENIX ATC'12*.