On the accuracy of trace replay methods for file system evaluation

Thiago Emmanuel Pereira, Livia Sampaio, Francisco Vilar Brasileiro
Systems and Computing Department
Universidade Federal de Campina Grande - Brazil
thiagoepdc@lsd.ufcg.edu.br, {livia, fubica}@computacao.ufcg.edu.br

***Abstract*—Current trace replay methods for file system evaluation fail to represent traced workloads accurately. When using a misrepresented workload one may take wrong conclusions about the system evaluation. For example, a system designer can miss performance problems if the replay of a trace produces an underloaded representation of the real workload. Even worse, one can take wrong design decisions, leading to optimization of untypical workloads. In this study, we captured and replayed traces from standard file systems using methods proposed in the literature, to exemplify the inaccuracy of state-of-art trace replay methods. We also exposed a shortcoming of current methodologies; in a replay of a general purpose workload trace, we observed a difference of up to 100% on request response time, caused by the choice of trace replay method.**

I. Introduction

Trace-based evaluation is a popular approach to assess the performance of file systems. However, up to now, there is no universally accepted methodology for replaying traces that are captured from real systems. Indeed, researchers have pointed out that results attained through trace-based analysis should be taken with concern, since it is believed that current approaches to reproduce traces do not represent the original workloads faithfully [1].

In this paper we provide further evidence that currently available approaches have severe limitations. Our study is based on comparing the results that were produced by different replay strategies proposed in the literature, when applied to traces collected from a standard NFS deployment, supporting the work of a typical research lab. Before presenting our experiments and the analysis of their results, we provide a brief description of trace-based evaluation methods.

II. Trace-based evaluation methods

Usually, each line of a file system trace stores data about a single request performed in the file system. Each request is associated with timestamps representing the time when the request was started and finished. The two main problems with the replay of a trace are related with the dependence that any two requests may have, which defines an order in which these requests should be replayed, and the exact timing at which the requests need to be replayed, when processing the trace.

Accurate representation of the workload stored in a trace requires deciding whether a workload is better represented by independent or by dependent requests (open and close models, respectively) [2]. Particularly, in the closed model, request arrivals are dependent on previous requests completion. For this reason, any change in the system under evaluation (for example, adopting a new design or using a faster device) that leads to different response times will also lead to a different workload. As pointed by Ganger and Patt [3], incorrect accounting of these feedback effects between request completion and arrivals cause I/O simulation methods to produce erroneous results.

Another difficulty concerns request timing decisions. More specifically, the problem is how to quantify the influence of the performance of the trace replay experimental environment on the inter-arrival time of dependent requests, considering that such inter-request intervals are a combination of computation and think[1] times. Whereas it is possible to adjust the computation time to reflect a faster hardware or a system optimization, the respective fractions of time devoted to computation and think times within the overall inter-request interval remain unknown. It is also unclear if application's think time is invariant with respect to system response time: would people behave differently if the system were slower or faster?

Most of the work in the literature that uses trace-based evaluation rely on *ad hoc* replayers [1]. A notable exception is the work by Zhu *et al.* that proposes the TBBT replayer [4]. The proposed replayer can be configured to work in four different modes, depending on the strategy used to order requests and to define the timing to execute them. Considering the workload dependency model, the TBBT replayer specifies two ordering policies: file system (FS) dependency and conservative. The FS dependency policy arranges requests based on file system modification semantics; for example, a request to write to a file cannot be dispatched before the request to create that file is dispatched. On the other hand, the conservative policy uses traced timestamps to define the order of requests, such that a request cannot be dispatched if any previous request (according to traced timestamps) is still executing. For the decision on timing, one possibility is to ignore any possible feedback caused by requests completion time and dispatch requests at full speed. Alternatively, a timestamp-based policy can be used, where inter-arrival times of replayed requests should be the same as the one recorded in the trace.

In this paper we have executed a number of experiments to

[1]Think time accounts for the time that the user takes to interact with the system.

assess the accuracy of the TBBT trace replayer methodology, considering all possible combinations of ordering and timing policies.

III. Materials and Methods

A. Trace capture method

We captured the activity of 15 NFS clients from our research laboratory during a continuous 3-week period, using a system call level approach. The NFS deployment corresponds to a general purpose, working group workload — load generated by productivity tools, programming, web browsing, music reproduction, etc.

B. Trace workload

Replaying the whole 3-week traces was infeasible. Instead, we selected a 10-minutes trace fragment from one of the 15 traced machines. This fragment is essentially composed by data operations, as shown in Figure 1.

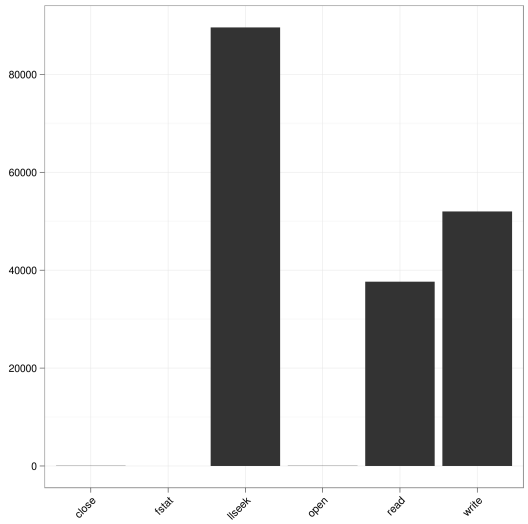


Figure 1: Distribution of requests according with its type.

Figure 2, depicts the distribution of trace fragment operations, in terms of requests per second. A major characteristic of this fragment, also found all along the 3-week capture, for all 15 traced clients, is the burst in the workload. In fact, as summarized in Table I, there is no activity during more than half of the 10-minutes interval (the request load median value is zero). As consequence, the inter-arrival intervals between requests vary wildly: while 90% of the intervals are smaller than 120 microseconds, the largest interval is greater than 1 minute.

	Reques Load			
Number of Requests	median	mean	max	CV
179379	0	321.5	17160	458.6

Table I: Summary of the selected 10-minutes fragment request load.

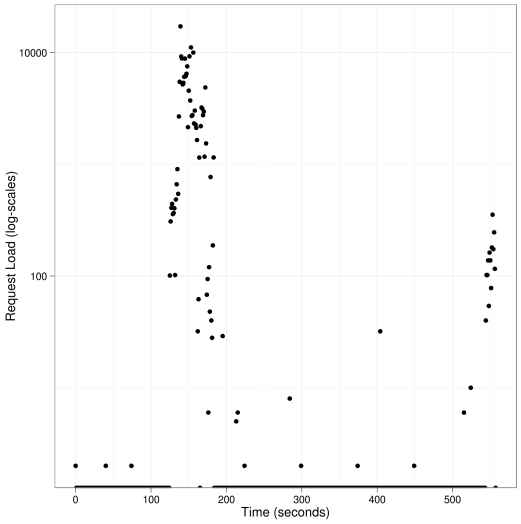


Figure 2: Request load of the selected 10-minutes interval.

Another important characteristic is that few processes (linux threads) concentrate most of file system activity. As pointed in Table II, there were 14 active processes during the 10-minutes interval. In particular, the three most demanding processes are responsible for more than 99% of the requests.

Number of processes	1st	2nd	3rd
14	98.86	1.05	0.04

Table II: Percentage of requests made by the 3 most active processes.

C. Trace replay method

Before the trace could be replayed, we recreated the file system state to reflect its contents at capture time: since we captured data from a production system, some files were accessed during trace capture but created before it, than we needed to recreate these files, otherwise some operations would fail. To help on reconstituting file system state, we created a snapshot of the traced file system content at the time of trace capture. The snapshot may be incomplete, for example, because of missing files not yet sent to permanent storage when the snapshot collector was running. To recreate these missed files, we had to assign sizes for them. We accomplished that by analyzing the trace and assigning the missing sizes to be equal to the largest offset manipulated for the respective files in the trace.

We released our trace replayer [2]. We plan to release the full 3-week trace at the SNIA trace repository [3] after anonymization.

D. Verification of trace replay accuracy

In this section, we verify whether we built the trace replayer rightly or not; if the trace replay does not proceed according

[2]https://github.com/thiagomanel/Beefs-trace-replayer/
[3]http://iotta.snia.org

to the order and timing characteristics dictated by the replay method of choice we cannot blame the method for the workload representation problems.

We applied different methods for verifying order and timing policies. Regarding order, we verify our replay tool by parsing a post-replay log against the expected order. Basically, the parsing program asserts that all dependent requests of a given request were executed after their predecessor. In its turn, timing policies were verified by contrasting the intended and actual request timestamps. Note that, as *fullspeed* policy does not restrict request timing, we do not analyze this policy.

There are two main sources of delay that contribute to timing inaccuracy in our replay tool: lock contention and the latency of OS sleeping utility.

Figures 3, and 4 illustrate the effects of these delays in a reproduction of two dependent requests, represented respectively by boxes 1 and 2 in Figure 3. In this scenario, request 1 starts at timestamp B1 and finishes at timestamp E1. Only after its end, request 2 is ready to be reproduced. However, it's not instantaneous: request 1 has to be sent to consume queue to signalize its termination, producer thread has to decide the next available requests and insert them into the produce queue, and finally a free consumer thread has to take request 2 from the produce queue. When all previous steps were done, at timestamp S, the consumer thread decides for how long it has to sleep — in our case *I_delay* time units — so that it matches the replay timing decision to start the request 2 at $B2$ timestamp. At this point, delay forces act again and the request 2 starts at timestamp $B'2$. The amount of time between the actual and and intended execution of a given request, $B'2-B2$, constitute the issue error E.

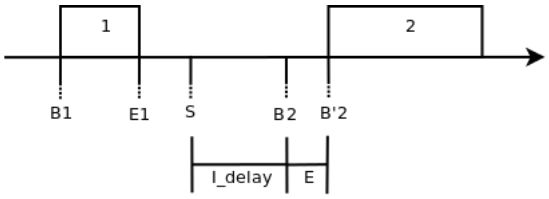


Figure 3: The intended beginning of request 2, $B2$, is postponed to timestamp $B'2$. The time interval between the actual and and intended execution of a given request, $B'2 - B2$, constitute the issue error E.

Figure 5 shows the empirical cumulative distribution function of the issue error found in the trace reproductions reported in Section IV. The median value for the issue error is 9 microseconds. Also, about 90% of replayed operations were affected by an issue error less 25 microseconds when using the FS policy, and 17 microseconds when using the conservative policy.

The consequences of trace replayer delays and its associated issue error must be analyzed in terms of how they modify workload characteristics; for slightly loaded workloads they might have a low impact. In this direction we evaluate the *compute time error ratio*: the relation between the issue error and interval between the end of a request and the start of its

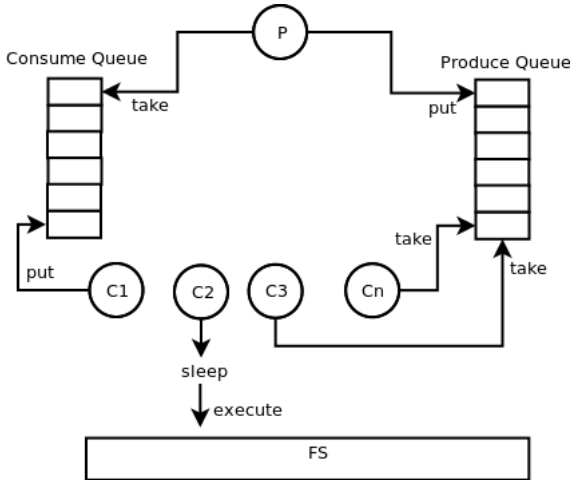


Figure 4: Trace replayer design. Consumer threads, $C1$ to C_n, take available requests from produce queue, then sleep for some time to match replay timing requirements, finally they execute requests in the filesystem F. After execution, consumer threads insert requests in consume queue. Producer thread takes consumed requests from consume queue to create new available requests based on replay order logic.

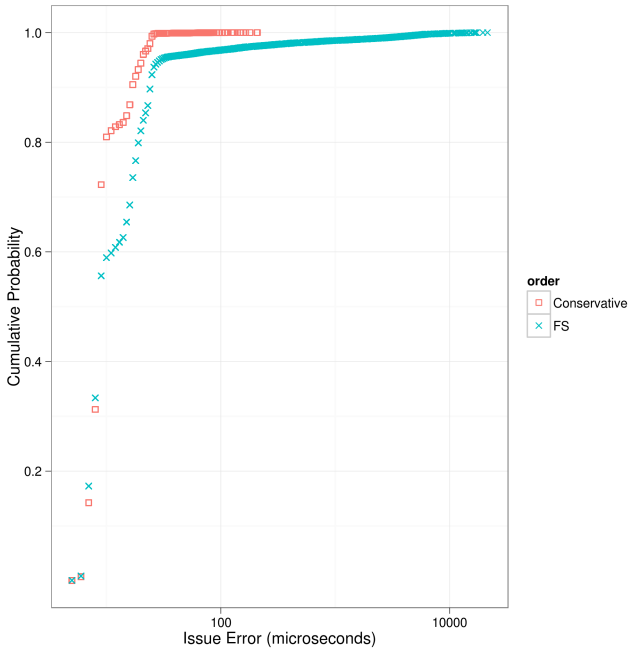


Figure 5: Empirical cumulative distribution function of issue error for requests in the replay of the selected 10-minutes trace fragment.

successor (the representation of compute time for dependent requests), thus $\frac{E}{B2-E1}$.

Figure 6 shows the empirical cumulative distribution function of compute time error ratio found in the trace reproductions of Section IV. This results indicate that, the issue error for the most of the requests inflate the emulated compute time

by less than 10%.

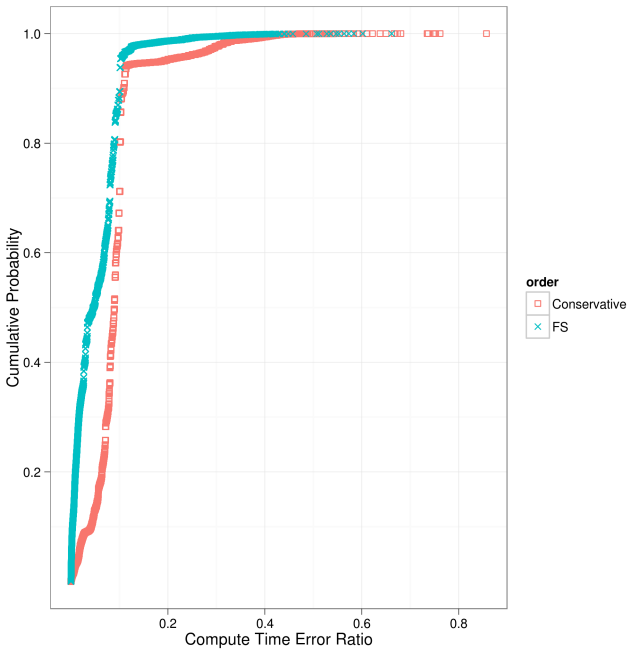


Figure 6: Empirical cumulative distribution function of compute time error ratio for the reproduction of the 10-minutes fragment.

To decide whether these errors have practical consequences or not, we performed a sensitivity analysis of issue error against our trace replay target metric, the response time. In this sensitivity analysis we created an additional, constant delay to each reproduced request and observed response time variation. Figure 7a shows the expected increment in the *compute time error ratio* as a result of the additional delay. Despite the raise of *compute time error ratio*, Figure 7b shows that there is no noticeable effect on response time.

E. Experimental Setup

Trace replay took place in the same desktop where we captured the trace that was replayed: an Intel $E6550$ Core 2 Duo 2.33GHz, with 1.92 GB of main memory, a 7200 RPM SATA disk with a 8 MB cache, running the Linux 2.6.32-42 kernel. Since the activity we captured was generated mainly by interactive applications, we used the ext4 file system — which is broadly deployed by Linux desktop users — to replay the trace.

F. Metrics

We analyzed our experiment in terms of the following metrics: (i) *request load*: the number of request arrivals per second; (ii) *request outdegree*: the number of dependent requests of a given request; and, (iii) *response time*: the time taken to process each request replayed.

These metrics serve two purposes. First, by analyzing *request outdegree* and *request load* we assess how different are workload representations provided by the known trace reproduction methods. Second, the *response time* metric gives us a direct test to whether the choice of replay method impacts performance evaluation or not.

IV. Evaluation

The choice of replay policy has a deep impact on the workload representation. Figure 8 compares *request load* under the combinations of timing and ordering policies against its traced, original workload. A trace replay combining timestamp timing and conservative policies takes approximately 10 minutes to be executed (the same from original workload); the same order police, when combined with fullspeed timing, takes less than 10 seconds to be executed. In its turn, the FS policy request load does not match the original workload when using the timestamp timing policy, and, more surprisingly takes longer to reproduce the trace fragment when using the fullspeed timing policy.

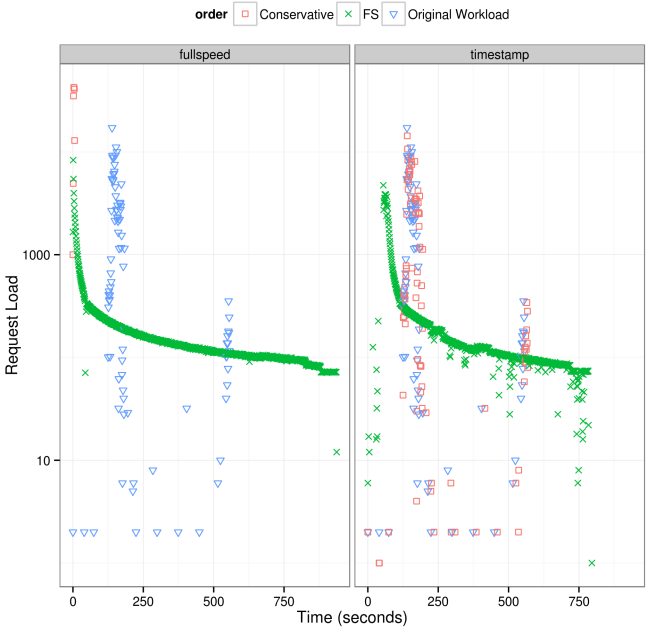
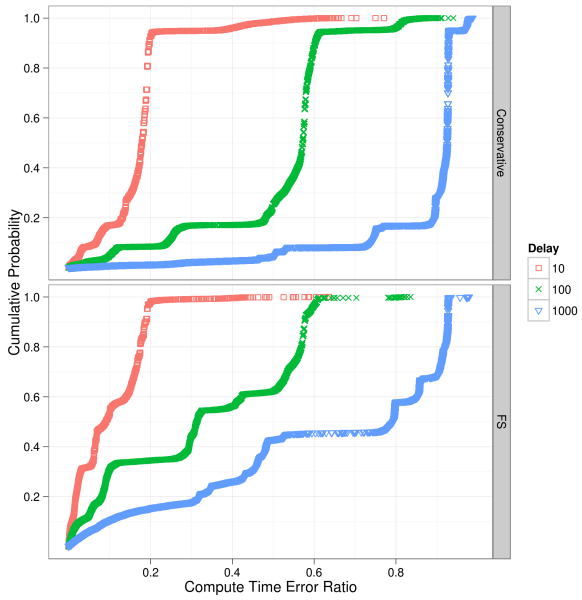


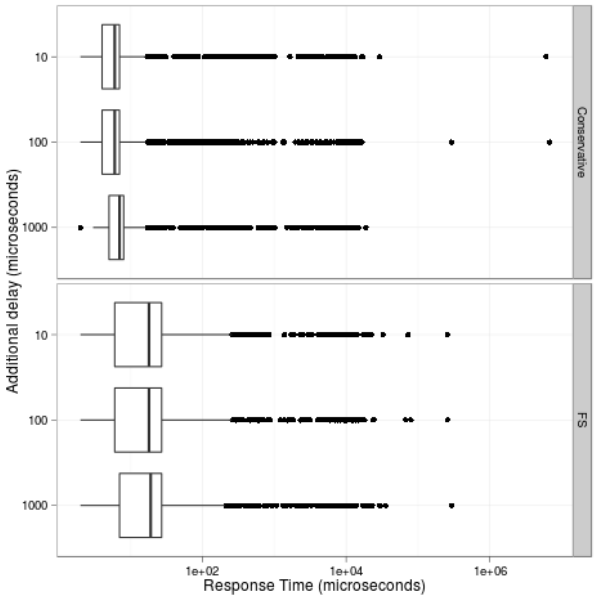
Figure 8: Request load for the replay of the selected 10-minutes trace fragment.

As expected, FS and conservative order policies also differ with regards to the *request outdegree* metric. While the maximum *request outdegree* for the conservative policy is 2, this metrics rises to 5 for the FS policy. In fact, as Table III shows, this contrast between FS and conservative policies is even higher for other trace fragment from different machines. Such huge variation endorses TBBT designers' conjectures about FS dependency policy being too aggressive.

Figure 9 shows the performance effects of choosing the specific and timing policies of trace replay. The two upper graphs report the empirical cumulative distribution function (ecdf) of response time for *read* requests in the selected 10 minutes trace fragment, while turn, the lower two graphs report the ecfd of response time for *write* requests.



(a) Empirical cumulative distribution function of compute time error ratio.



(b) Request response time.

Figure 7: Sensitivity analysis of issue error effect on reproduced requests response time. Additional delay ranges from 10 to 1000 microseconds.

conservative (median, max)	FS dependency (median, max)
$(1, 2)$	$(1, 1909)$
$(1, 3)$	$(1, 376)$
$(1, 5)$	$(2, 834)$
(1, 12)	**(2, 51160)**
$(1, 5)$	$(1, 114)$
$(1, 4)$	$(2, 4012)$
$(1, 3)$	$(1, 171)$
$(1, 2)$	$(1, 129)$

Table III: Median and maximal request outdegree for trace fragments considering conservative and FS policies.

We observe that **the choice of ordering policy, conservative or FS dependency, causes a difference up to 100%** in most of the percentile ranges we report, no matter which request operation type (*read* or *write*) is considered. On the other hand, **there is no noticeable effect of the timing policy choice.**

FS policy slowdown is a consequence of its workload misrepresentation, more specifically, its artificial raise of concurrency. Figure 10 correlates the response time of replayed requests with the number of concurrent requests running at the moment. We observe that the FS policy deviates the response typical values from about 4 microseconds to 10 microseconds, and this deviation happens when the number of concurrent request grows to 2.

V. Conclusions and Future Work

This paper supports file system community claims over the (lack of) quality of file system performance evaluation methods. In this direction, we provided quantitative evidence that known trace replay methods are inaccurate — in our trace replay experiments we observed up to 100% on request response time, caused by the choice of trace replay method.

We believe that such large methodological errors do have practical consequences. For example, a programmer when using a trace replay to profile its system may stop tunning earlier if he meets (erroneously) the target performance requirement.

By illustrating the pitfalls of current methods, we expected that we can devise guidelines on how to choose the most suitable method to a given situation. Also, it can help us to take informed decision on how to improve the replay methodology. The results we show in this paper are a first step to this end; we confirmed that TBBT FS policy is inaccurate. We also suggested that we need finer grain dependency analysis.

File system literature points a number of claims that have yet to be proven. For example, Zhu et al. [4] advised that their conservative ordering policy may be sensitive to latency variations in the system under test. Is there any outcome if the system under test is slower or faster than the system under capture ? Another open problem that we introduced in section II regards proper accounting for compute and think-time; they are oblivious after trace capture.

References

[1] A. Traeger, E. Zadok, N. Joukov, and C. Wright, "A nine year study of file system and storage benchmarking," *ACM Trans. on Storage (TOS)*, vol. 4, no. 2, pp. 1–56, May 2008.

[2] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: a cautionary tale," in *Proc. of the 3rd conference on Networked Systems Design & Implementation (NSDI'06)*, San Jose, USA, May 2006.

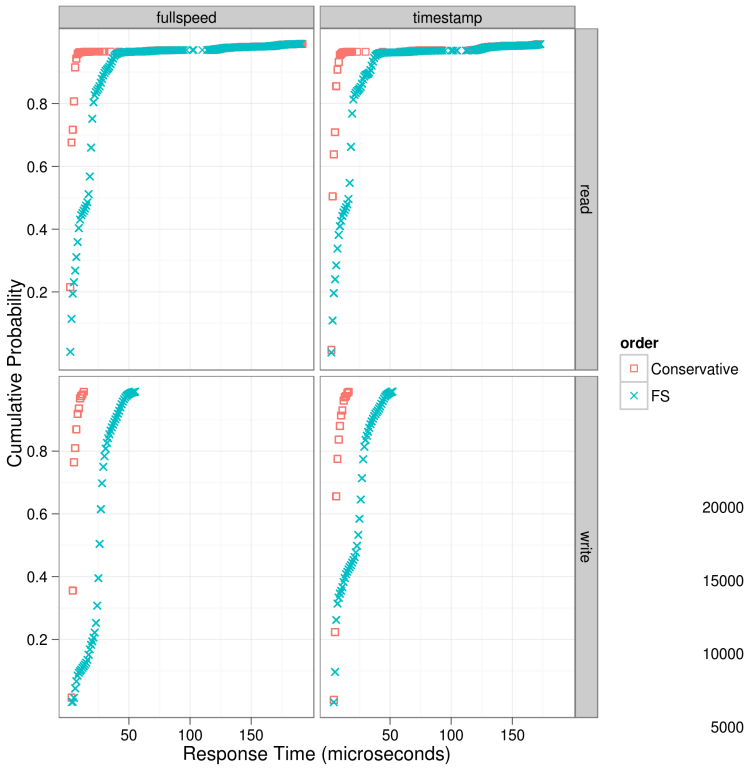


Figure 9: Empirical cumulative distribution function of response time for *read* and *write* requests in the replay of the selected 10-minutes trace fragment. Ecdf values range up to the 99th percentile.

[3] G. R. Ganger and Y. N. Patt, "Using system-level models to evaluate i/o subsystem designs," *IEEE Trans. Comput.*, vol. 47, pp. 667–678, Jun. 1998.

[4] N. Zhu, J. Chen, and T.-C. Chiueh, "Tbbt: scalable and accurate trace replay for file server evaluation," in *Proc. of the 4th USENIX Conference on File and Storage Technologies (FAST'05)*, San Francisco, USA, Dec. 2005, pp. 24–24.

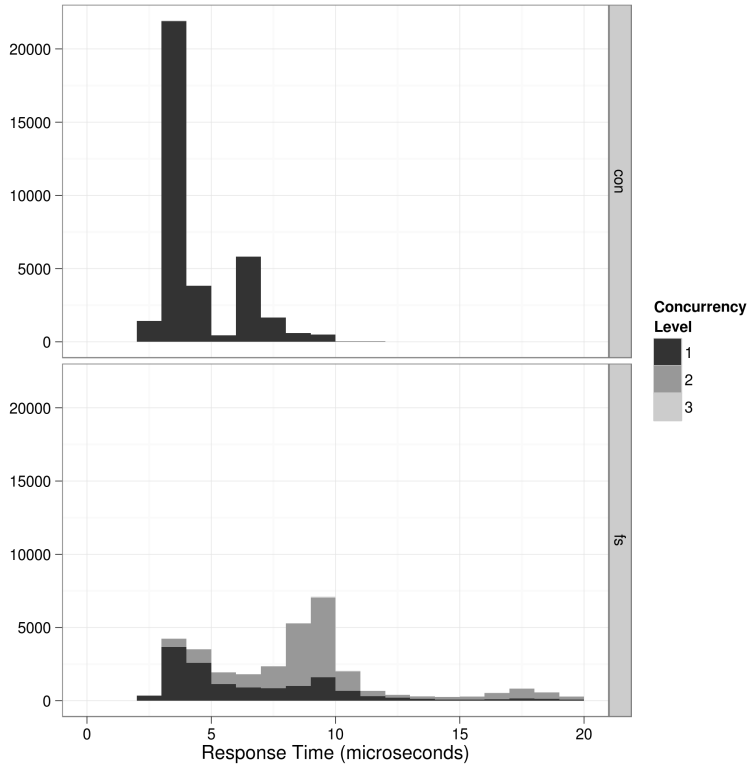


Figure 10: Histogram for request response time according to the number of concurrent requests.