Evaluating Scalability Bottlenecks by Workload Extrapolation

Rong Shi, Yifan Gan, Yang Wang
Dept. of Computer Science and Engineering
The Ohio State University
{shi.268, gan.101, wang.7564}@osu.edu

***Abstract*—Testing a scalability bottleneck requires a large system to generate sufficient load, which is usually not accessible to researchers. To address this problem, this paper extrapolates the workload to a bottleneck node. The key observation that motivates our approach is that systems at a large scale are often repeating their behaviors at small scales, by running a job more times, running more nodes of the same type, or running more iterations of the same loop. Following this observation, we record a node's workloads at small scales and extrapolate such workload at a large scale. Towards this goal, we have developed PatternMiner, a semi-automatic tool to identify how workload patterns change with scale.**

We have tested our method on HDFS NameNode and YARN's Resource Manager. Our evaluation shows that PatternMiner is able to predict 98% of the workloads for NameNode and 83% of the workloads for the Resource Manager. Furthermore, by utilizing the extrapolated workload, we are able to emulate a cluster of up to 60,000 nodes with only 8 physical machines to evaluate NameNode and Resource Manager.

I. Introduction

To test centralized scalability bottlenecks with limited resource, this paper proposes a semi-automatic approach to synthesize a workload that resembles the one the bottleneck would observe in a large-scale deployment.

To simplify design, modern large-scale distributed systems usually incorporate a few centralized metadata servers (e.g. NameNode in HDFS [53]) to provide a global view of the whole system [5, 7, 11, 17, 24, 29, 51, 53, 56]. It is well-known that these centralized servers will eventually become bottlenecks as system scale increases [20, 52, 58]. Therefore, to understand and improve system scalability, it is critical to investigate the performance of these bottlenecks.

The traditional approach to achieve this goal is to run a benchmark [30, 42, 47, 55] or play a trace [13, 33, 39, 61] to test the target system. When testing scalability bottlenecks, however, we face a new challenge: being aware of the drawback of centralized servers, large distributed systems make efforts to minimize their load. As a result, to generate sufficient load to a centralized server, we need to deploy a large system, which requires at least thousands of machines [52]. Even with the help of public testbeds [10, 15] and clouds [4, 38], getting hundreds of nodes is expensive or requires a long time to schedule, not to mention thousands or more nodes. Even industrial researchers are not immune to this problem, because their clusters are a primary source of revenue. For example, Facebook chooses to test their systems in production [57], which reduces the cost but incurs the risk of an unknown problem striking the production cluster.

To address this problem, previous works have made attempts in one of two ways. The first is to run multiple small to medium-scale experiments, measure the resource consumption of bottleneck nodes in these experiments, and extrapolate such consumption to large scale [17, 44, 59, 64]. While this approach works well in some cases, its fundamental assumption—resource consumption of bottleneck nodes increases linearly, or at least predictably, with the scale of the system—does not always hold. For example, performance collapse is frequently reported when the system is under heavy load [8, 37, 50, 58], indicating resource consumption can have sudden changes at certain point.

The second approach is to synthesize a workload that resembles the one the bottleneck would observe in a real benchmark. Such synthetic workload can be generated either by relying on developer's knowledge or by replacing other nodes with *stubs*, which can mimic the behavior of a real node with less overhead. However, these approaches only work well with simple benchmarks (e.g. read or write files) or under specific assumptions (e.g. content of data does not affect processing [2, 35, 58]): with a more sophisticated benchmark, the complexity quickly grows. For example, when we tried to synthesize a MapReduce workload to test HDFS NameNode, even the simplest benchmark (i.e. WordCount) on a small dataset (i.e. 5GB) generates a total number of 1171 remote procedure calls (RPCs) with 23592 arguments.

The goal of our work is to automate the process of synthesizing a workload. The key observation that motivates our approach is that nodes' behaviors in a large-scale deployment, to a large degree, are a repetition of their behaviors in a small-scale deployment. Such repetition is due to running the same job multiple times [1, 3, 18], running similar tasks on multiple nodes (e.g. Mapper and Reducer in MapReduce), and running multiple iterations of the same loop. Following this observation, our approach tests the target system with real nodes under different (small to medium) scales, records their traces, analyzes how such traces change with scale, extrapolates such traces at a larger scale, and finally plays the extrapolated trace to the bottleneck node.

We have developed PatternMiner, a semi-automatic tool to help developers identify important patterns in workload logs. It has three key functions:

- Identify static and repeated behavior. By utilizing sequence alignment techniques [40], PatternMiner can identify which part of the trace remains static across experiments of different scales, which part is repeated more times in larger experiments, and which part changes irregularly, requiring the developer's effort.
- Identify patterns for arguments values. For the system to work correctly, argument values of different requests must meet a variety of constraints. PatternMiner first identifies whether the value of an argument follows a predefined set of patterns, such as constant or iteration number. If not, PatternMiner further searches whether the value is retrieved from previous requests.
- Summarize patterns. For found repeated patterns, PatternMiner summarizes their rules of repetition, which allows us to validate patterns across experiments and extrapolate patterns at a larger scale.

We have applied our method to HDFS NameNode [53] and YARN's Resource Manager [56]. The evaluation confirms the feasibility of workload extrapolation: PatternMiner can predict 98% of the traces for the NameNode and 83% of the traces for the Resource Manager. Among the remaining ones, most can be addressed by a few lines of code and only two events require a considerable amount of developer's effort. We are able to partially validate the accuracy of our approach by comparing our extrapolated workloads to those from a real 500-node experiment on Microsoft Azure.

By utilizing the extrapolated workload, we are able to emulate a cluster of up to 60,000 nodes with only 8 physical machines to stress test the NameNode and the Resource Manager. This emulation allows developers to know the maximum scale of the system with limited resources. Also, we find some problems that only happen at large scales and we are able to confirm those problems.

II. Motivation and Challenges

Our approach is motivated by the observation that a distributed system at a large scale is often repeating its behavior at small scales. Such repetition can happen at different levels:

- At the highest level, as reported in multiple studies [1, 3, 18], users tend to run the same job many times, often on different inputs.
- Inside a job, different nodes may execute similar tasks. Such repetition can be observed in both the design of these systems and the traces we collected (see Section V for details). For example, in Hadoop MapReduce, we find Mappers and Reducers are sending similar sequence of requests to NameNode respectively; in Spark, we find the Application Master and Node Managers are sending similar sequence of requests to the Resource Manager respectively; in HBase, we find the Region Servers are sending similar sequence of requests to ZooKeeper. Such repetition can also be found in MPI-based parallel computation applications [46, 48, 54].
- At the lowest level, a task often uses loops to handle given data. For example, a Mapper uses a loop to read all data of the input file; a Reducer uses a loop to output data. Multiple iterations of the same loop can also generate similar sequence of requests.

Essentially, such repetition is rooted in the basic principle of software engineering: developers tend to design their code to be *reusable* across iterations, across nodes, and across jobs. As a result, such pieces of code are executed multiple times at runtime and naturally leads to a repetition of behavior at runtime.

Such repetition provides us with an opportunity to extrapolate a system's behavior at a large scale, by analyzing its behavior at small scales. Applying this idea to a complicated system, however, is challenging. To motivate our work, we first show our experience of manually analyzing the workload for HDFS NameNode.

HDFS stores data on a number of DataNodes and stores metadata, such as directory structure and locations of data blocks, on a single (maybe replicated) NameNode. We run MapReduce WordCount, which is one of the simplest benchmarks, over HDFS and collect all RPCs to NameNode. Following are our observations:

- Because of the size of the traces, it is difficult to manually identify what pattern is repeated. Even when running on a small dataset (5GB), the benchmark generates a complicated trace: it contains a total number of 1171 RPC instances, and a total number of 23592 arguments in these RPCs. They include 21 types of RPCs and 150 types of arguments from six types of nodes. With our tool, we find several types of repeated sequences and one of them is a sequence of 16 RPCs to upload a file: manually identifying such a long pattern inside a long trace is hard.
- A pattern occurring multiple times does not mean all its occurrences are identical. Still taking MapReduce as an example, each Mapper needs to open a file, read it, and finally close it, but obviously each Mapper needs to open a different file. This means we cannot simply copy paste the repeated pattern. Furthermore, there exist complicated dependencies among one pattern (e.g. the Mapper can only close the file it opened): violating such dependencies may cause the NameNode to throw exceptions and fail the test.
- There exist patterns that are hard for the computer to analyze automatically: this includes non-deterministic RPCs and RPC arguments with no clear pattern (e.g. port number). Many of them can be easily addressed by developers who have knowledge of the system.

These observations motivate us to develop a semi-automatic tool: it tries its best to extract patterns from workload traces; for those it cannot find an appropriate pattern, it presents a concise summary to the developer.

III. Design

As shown in Figure 1, workload extrapolation works in three phases: in the preparing phase, we run the real system and benchmark with multiple small to medium experiments and record the workload traces (input requests) to the target bottleneck node. In the mining phase, we analyze patterns

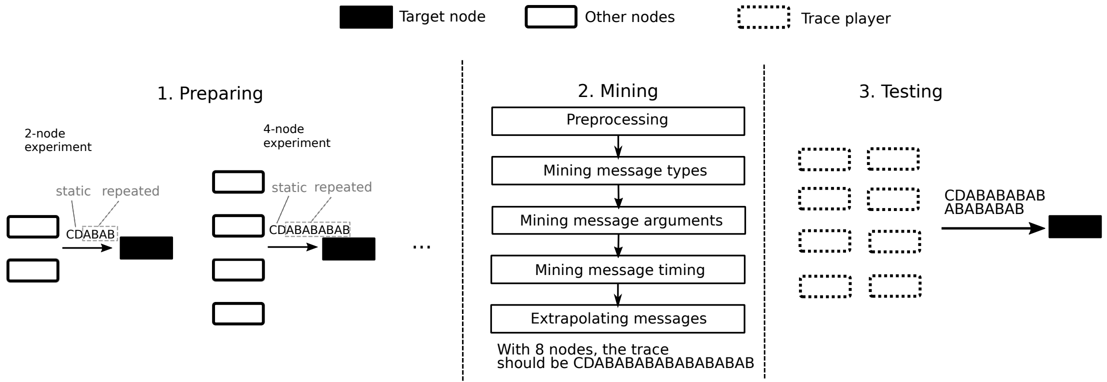

Fig. 1. Work flow of workload extrapolation

in these collected traces and extrapolate how the trace looks like at a larger scale. Towards this goal, we have developed PatternMiner, a tool to help developers extract such patterns. In the testing phase, we play the extrapolated trace to the target bottleneck node to measure its performance.

Next we present each step in detail. Note that our approach is semi-automatic: it asks the developers if it finds an unknown pattern.

A. Preparing

In this phase, we run the benchmark on the real system with different small to medium-scale experiments and log all input and output messages of the target node.

Although it is possible to automatically instrument the target system for logging, our current implementation simply requires developers to implement the logging function, because most systems already have such functions and slightly modifying them can achieve our goal.

Such logs should contain complete information of the corresponding messages. Furthermore, to facilitate analysis in the next phase, such logs should be semantically meaningful, providing type information of data in such messages. For services implemented under the remote procedure call (RPC) model, the RPC layer is a natural place to log traces. For systems using message passing, our approach can record raw network packets but it expects the developer to provide functions to deserialize raw network packets into semantically meaningful variables: such functions usually already exist in the target system. Automatically extracting semantically meaningful information from raw bytes is out of the scope of this paper. For simplicity, we will use the RPC model in this paper, because with deserialization functions, message passing is fundamentally no different from RPC.

PatternMiner in the second phase expects the following trace format: for each RPC, the trace should contain its timestamp, a sender ID to identify the node that issues the RPC (a thread ID if the node uses multi-threading [65]), the RPC function name, and the types and values of all its arguments and return. Table I gives an example of the trace format of the *mkdirs* RPC and its arguments.

TABLE I
EXAMPLE OF PARSED TRACE FORMAT. REQ/RESP = REQUEST/RESPONSE ARGUMENTS (LIST OF {TYPE: VALUE} PAIRS)

time_stamp	sender_id	RPC	REQ	RESP
19:04:01.8	11394:74	mkdirs	{*src*: "/in/_temp/1", *masked.perm*: 493, *createParent*: true}	{*result*: true}

B. Mining

In the mining phase, we analyze the recorded traces with PatternMiner, and predict how the traces look like at a larger scale. Since we assume certain patterns of RPCs will be repeated across experiments, the key goal of PatternMiner is to find which patterns are repeated (called "repeated pattern" in the rest of this paper), which patterns are static (called "static pattern" in the rest of this paper), and which patterns are irregular, requiring developers' effort. Given such information, we can then predict how the traces look like at a large scale for the purpose of evaluation in the third phase. To be concrete, we need to predict the types (i.e. function names) of all RPCs, values of all their arguments, and their timestamps. We don't need to predict their return values because they will be given by the target node at runtime.

PatternMiner relies on existing data mining techniques to find patterns, but it faces two challenges: first, PatternMiner not only needs to identify patterns in each experiment, but also needs to find how such patterns change across scale. Second, patterns found in one experiment may not hold across experiments, so PatternMiner needs to validate such patterns across experiments. Both challenges require PatternMiner to compare found patterns across different experiments, but repeated patterns are often not directly comparable since they have different lengths across experiments of different scales. To solve this problem, PatternMiner's *summarizes repeated patterns* by extracting their rules of repetition: these rules are directly comparable across experiments.

*1) **Preprocessing**:* PatternMiner first separates the raw log, which contains all RPCs collected at the target node, into separate logs based on the sender of each RPC. PatternMiner

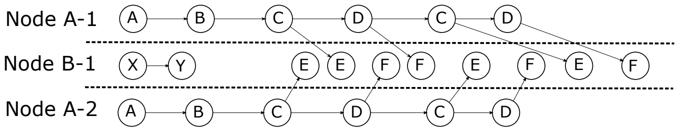

Fig. 2. Relocate RPCs based on causal ordering

```
register("10.0.0.1")
list("/data/input");
100 = create("/data/input/input0");
write(100, 1024)
close(100)
105 = create("/data/input/input1");
write(105, 1024)
close(105)
107 = create("/data/input/input2");
write(107, 1024)
close(107)
```

Fig. 3. An example of a log of RPCs

separates logs first because, as shown in Section 2, the repetition of patterns comes from running nodes of the same type and running iterations of the same loop: for both cases, it is easier to extract such patterns from separated logs.

Then PatternMiner relocates part of RPCs based on causal ordering. The purpose of this step is to reduce nondeterminism incurred by parallelism. For example, as shown in Figure 2, RPCs E and F are in the log of node B-1, but they are actually triggered by RPCs C and D from two concurrent nodes A-1 and A-2. Since nodes A-1 and A-2 are both repeatedly calling C and D, E and F are repeatedly triggered, but if we look at the log of node B-1, such repetition may be obscured by nondeterminism in timing: in this example, node B-1 logs EEFFEFEF, in which the repetition of EF is not clear. PatternMiner relies on causal logging technique [14, 36, 65], which can identify the causal relationship of different events, to alleviate the problem: if causal logging finds one RPC r1 is triggered by another RPC r2 from node n, PatternMiner relocates r1 to n's log and put it after r2. In this example, PatternMiner will relocate E and F to nodes A-1 and A-2, so that the logs of A-1 and A-2 become ABCEDFCEDF, in which one can observe the repetition of CEDF. If r1 can trigger other RPCs, PatternMiner performs relocation recursively. In our current implementation, we track causal relationship in logging by tracking unique IDs of certain tasks.

Then PatternMiner tries to identify patterns that are repeated across different nodes: it clusters logs with similar histogram of RPC function names into the same type, since they are probably from the same type of nodes. For example, in Figure 2, nodes A-1 and A-2 will be clustered into the same type. PatternMiner will ask the developer to verify its clustering result.

*2) **Mining patterns for RPC types:*** Then PatternMiner tries to extract repeated and static patterns inside each node's log. PatternMiner first focuses on the RPC types, which are represented by the function names of the RPCs. To illustrate the procedure, we use an example as shown in Figure 3. In this phase, PatternMiner focuses on the sequence of their types [**register, list, create, write, close, create, write, close, create, write, close**] to see if there exists any repeated pattern.

Repetition of patterns can occur in both real-time field (e.g. heartbeat) and logical-time field (e.g. loop) and these two often interleave with each other. PatternMiner first extracts repeated patterns from real-time field since they are usually simpler. PatternMiner identifies a certain type of RPCs as real-time repeated if the real-time interval of their occurrences is regular.

Then PatternMiner attempts to separate static and repeated patterns in the logical-time field.

Detect nondeterministic RPCs: Nondeterministic RPCs will create a noise for extracting patterns. PatternMiner first tries to identify nondeterministic RPCs by comparing logs from the same type of nodes and by comparing logs from experiments with the same scale. PatternMiner uses standard text comparison technique to "diff" the sequence of RPC names in those logs and present the different part to the developers to ask how to handle such nondeterministic RPCs. Of course, PatternMiner only works well if there are not many of such nondeterministic RPCs. Otherwise, either the developers need a lot of effort or the system simply becomes unpredictable.

Identify static and repeated patterns: Then PatternMiner separates static and repeated patterns in the logs.

PatternMiner first tries to identify all patterns that are repeated more than once in each node's log (Algorithm 1). It is a brute-force search that iterates all sub-sequences of the log and counts how many times they are repeated consecutively. Since multiple repeated segments may overlap, PatternMiner gives a preference to those with a higher coverage. The complexity of this algorithm is $O(N^3)$, in which N is the number of RPCs in one log. Although we find it to be acceptable in our small-scale experiments, it is possible to incorporate optimized versions [21, 40] if the log is long.

After this step, each node's log is naturally separated into multiple segments. PatternMiner summarizes each segment into a template $< type, pattern, repetition >$: "type" is either *repeated* or *static*; for a static segment, "pattern" records all RPCs in the segment and "repetition" is 1; for a repeated segment, "pattern" records the RPC sequence that is repeated and "repetition" records the number of repetition. Taking traces in Figure 3 as an example, after this step, PatternMiner can identify two segments: the first is $< static, [\textbf{register}, \textbf{list}], 1 >$ and the second is $< repeated, [\textbf{create}, \textbf{write}, \textbf{close}], 3 >$.

Cross validation: Then PatternMiner validates its key assumption by checking the following condition: for logs from the same type of nodes but from experiments of different scales, their segment information should be the same, except that for repeated segments, the number of repetition could be different. If validation succeeds, PatternMiner will move to the next phase. Otherwise, PatternMiner reports to the developer. There is one exception in this case: it is possible that a pattern is actually repeated, but it occurs only once in a small experiment and thus is classified as static. By comparing results across experiments, PatternMiner adjusts it to repeated.

Algorithm 1: Search for repeated segments in an RPC sequence

```
Input: RPC sequence
Output: list of repeated segments
$list = []$
for $chunk\_sz \leftarrow 1$ to $seq\_len/2$ do
    for $offset \leftarrow 0$ to $chunk\_sz$ do
        split RPC sequence into multiple chunks: chunk0 starts at offset; chunk1 starts at offset+chunk_sz, ...
        check if multiple consecutive chunks are the same: if so, record the chunk and its number of repetition in $list$;
        filter out duplicate patterns with either equal or smaller coverage;
sort $list$ by descending order of coverage ($len(sub\_seq) \times freq$), then increasing order of $len(sub\_seq)$
$result$ = []
while true do
    find the top segment from $list$ that does not overlap with any one in $result$
    if not found then
        break
    else
        add the segment to $list$
return $result$
```

Compute number of repetition: Finally PatternMiner tries to identify that, for repeated patterns, how the number of repetition changes with scale. PatternMiner uses linear regression to extract this relationship. If the number of repetition is constant, PatternMiner adjusts the type of the segment as "static".

3) Mining patterns for RPC arguments: To evaluate the target node, we not only need to extrapolate the types of RPCs, but also need to extrapolate their argument values. This is critical for the validity of the extrapolated trace, because the target node may rely on invariants in these arguments. Taking Figure 3 as an example, if a node wants to write to a file, this file must have already been created, and thus the **write** RPC should use the same file ID as the one returned by **create**. This means that it is impossible to predict argument values by looking at an isolated RPC.

Our approach simplifies this task in two ways: first, PatternMiner can gain information from real traces of small-scale experiments. For example, in MapReduce, a Mapper only needs read permission when opening a file, and such rules can be extracted from small-scale experiments. Second, segment information we gain in the previous phase can serve as an accurate context information for an RPC. In particular, for each RPC instance, PatternMiner computes which type of node it belongs to, which segment it belongs to, and its offset in the segment. If the corresponding segment is a repeated one, PatternMiner further computes which iteration it belongs to and its offset in the iteration. Such segment information, together with the environment information such as the configuration, form the *context* of the RPC. Not surprisingly, we observe RPCs with similar context to have significant similarity, which allows us to predict their argument values.

In practice, we find while many arguments' values can be determined by solely looking at the context of the corresponding RPC, others' values can be from the values of arguments or returns of previous RPC calls. We call the first type *regular pattern* and the second type *information flow* in this paper. Since regular pattern is easier to extract than information flow, PatternMiner first tries to check if arguments can match with any predefined regular patterns, and for those which do not, PatternMiner analyzes its information flow.

Regular pattern: An argument's value follows a regular pattern if its value can be solely determined by its context (e.g. segment information, configuration, etc). Typical examples include constant values, values that are regularly changed across iterations, etc. PatternMiner uses a three-phase algorithm to determine if an argument's value follows a regular pattern.

In the first phase, PatternMiner performs *cross-iteration summarization*: for each repeated segment, PatternMiner compares arguments of RPCs with the same offset in iteration but with different iteration number. For each argument, PatternMiner performs a *diff* operation to its values across different iterations: if there is no difference, PatternMiner classifies the argument as a constant value. Otherwise, PatternMiner checks whether the different part matches with a predefined set of patterns, such as linear relationship with iteration number or a matching with any other variables in its context. If a match is found, PatternMiner replaces the actual value with a template. Taking Figure 3 as an example, for the repeated segment **[create, write, close]**, PatternMiner will identify the argument of **create** to follow a regular pattern "/data/input/input[iteration]" and the second argument of **write** to follow a constant pattern 1024. For arguments that do not match any predefined patterns, PatternMiner marks them as "unknown" (e.g. the first argument of **write** and the argument of **close**). At the end of this phase, PatternMiner summarizes each repeated segment into only one iteration, which is called a *representative iteration* of the repeated segment. For example, the representative iteration of Figure 3 is [**create**(/data/input/input[iteration]), **write**(unknown,1024), **close**(unknown)].

In the second phase, PatternMiner performs *cross-node summarization* by investigating logs from the same type of node. For arguments with the same context in different nodes, PatternMiner uses the same algorithm as in the first phase to check if they follow any regular patterns. Note that patterns found in the first phase may not hold in the second phase: for example, one argument may have constant value in one node, but may not have the same value across different nodes. PatternMiner reclassifies such patterns. At the end of this phase, PatternMiner summarize each type of nodes into only one node, which is called a *representative node*.

After the above two phases, each experiment should have the same number of representative nodes, and same type of representative nodes from different experiments should have the same number of RPCs and arguments. This allows PatternMiner to perform *cross-experiment validation* in the third phase: PatternMiner checks that, for RPCs from different experiments but with the same context, their arguments should follow the same pattern. If so, PatternMiner records the pattern. For the remaining ones, PatternMiner performs information flow analysis.

Information flow: Some arguments' values are the same as those of arguments or return values of previous RPCs, which means there is an information flow among multiple RPCs. PatternMiner tries to find such relationship by searching back from the target RPC to see if any previous RPC's argument or return value has the same value: if so, PatternMiner marks the previous RPC as the *parent* and the target RPC as the *child*. Note that for relocated RPCs (Figure 2), PatternMiner searches in both the original component of the RPC and the component it is relocated to. Then PatternMiner summarizes the pattern it finds based on the locations of parent and child:

- Both parent and child are in static segments. This is the simplest case. PatternMiner simply records the parent's location and the child's location.
- Parent is in a static segment and child is in a repeated segment. PatternMiner checks whether RPCs from all iterations have the same parent. If so, PatternMiner records the parent's location and marks all corresponding RPCs in the repeated segment as children. Otherwise, PatternMiner reports to the developer.
- Parent is in a repeated segment and child is in a static one. PatternMiner checks whether the iteration number of the parent follows a specific pattern (e.g. constant or last iteration). Otherwise, PatternMiner reports to the developer.
- Both parent and child are in repeated segments. PatternMiner checks whether the pair of (parent iteration number, child iteration number) follows a specific pattern like (i, i+offset). Otherwise, PatternMiner reports to the developer.

Taking Figure 3 as an example, PatternMiner will identify that the second argument of **write** is from the return value of **create**, and the argument of **close** is from the second argument of **write**. Furthermore, PatternMiner identifies that for all three instances of such relationship, both parent and child are in a repeated segment and have the same iteration number, so PatternMiner summarizes the relationship as (iter[i].rpc[0].return → iter[i].rpc[1].args[0]) and (iter[i].rpc[1].args[0] → iter[i].rpc[2].args[0]). Note that PatternMiner does not need to predict return values because they will be given by the target node at runtime.

After summarization, PatternMiner validates the results across different nodes of the same type, and same type of nodes from different experiments. If any mismatch is detected, PatternMiner reports to the developer.

4) ***Mining patterns for timing:*** In the final phase, PatternMiner identifies patterns for timing. PatternMiner tries to extrapolate two kinds of timings: one is the time interval between two consecutive RPCs from the same node; the other is the starting time of each node.

To extrapolate the time interval between two consecutive RPCs, PatternMiner computes the corresponding time interval in small-scale experiments and use linear regression to check how it changes with scale.

The starting time of a node often depends on the progress of other nodes. For example, in MapReduce, a Reducer starts after all Mappers finish. PatternMiner predefines a set of patterns and check whether such patterns exist in small scale experiment: for example, the *fork* pattern means a number of nodes will start after a node finishes; the *join* pattern means a node will start after a number of nodes finishes; etc. Once again, PatternMiner will ask developers to verify the result because developers usually have a good understanding.

5) ***Extrapolating traces at large scales:*** In this stage, PatternMiner uses all the information it gains from previous analysis to extrapolate traces at a large scale. Similar as the analysis stage, PatternMiner performs extrapolation in several steps.

Generate the final setting: This step requires the developers' help: the developer needs to specify the configuration, in particular the number of each type of nodes. PatternMiner has no knowledge about such setting, so it has to rely on the developer's knowledge.

Extrapolate RPC types for each type of node: PatternMiner uses segment information to generate the sequence of types of RPCs for each node. For each repeated pattern, PatternMiner predicts its number of iterations based on the setting of the experiment. Real-time related patterns are not generated in this step, but generated at runtime, because it needs real-time information.

Extrapolate RPC arguments: For arguments that follow regular patterns, PatternMiner will directly fill its values in this stage. For arguments that get values from previous RPCs, PatternMiner will fill a template in this stage, indicating where to retrieve the information. Such information will be retrieved at runtime.

Extrapolate timing of RPCs: To predict intervals of consecutive RPCs, PatternMiner uses the extracted information directly. To simulate such intervals, PatternMiner inserts *sleep()* call before corresponding RPCs. To accelerate long experiments, PatternMiner provides a tunable parameter to shorten or eliminate such intervals. To predict when a node starts, PatternMiner generates a number of rules like "node i should start after node j and node k complete".

C. Testing

Finally, we play the extrapolated trace to test the target node. Figure 4 presents the design of this phase. The key components of this step are a number of node players that replace the real nodes to generate RPCs to the target node. Internally, each player is composed of three components: an interpreter to read the pattern generated by PatternMiner and to convert them into actual RPCs; an executor to make the RPCs and to receive the replies from the target node; and a recorder to

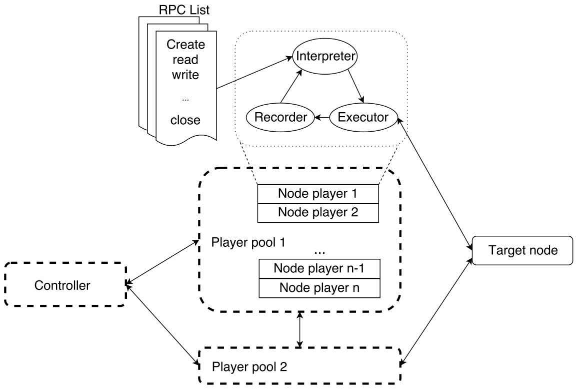

Fig. 4. Play the extrapolated workload to test target node

record all past RPCs and replies, which may be used by the interpreter later. Besides, we create a centralized controller to coordinate all players: it uses timing information extracted by the PatternMiner to decide when to start a player.

IV. Limitation and Future Work

First, our approach requires running experiments with different scales, which is feasible for benchmark testing, but is usually not feasible in a production system. Since benchmark testing is widely used for performance comparison and for identifying problems before deploying a system, our approach is still beneficial in multiple ways. If industry can provide production traces at different scales, our approach may generate a larger trace.

The other limitations of our approach come from its assumptions about the target system. In this section, we summarize its assumptions and try to relax them.

The key assumption is that system at a large scale is repeating its behavior at small scales. If the target system runs different algorithms or different types of nodes at a specific scale, then of course our approach does not work. In practice, behaviors like load imbalance, failure recovery, and straggler mitigation are often more likely to occur at larger scales: if these behaviors are not triggered in the preparing phase, then of course our approach will not be able to predict them. However, if developers have knowledge about such behaviors (e.g. the distribution of load and the frequency of failures), they may deliberately trigger them in the preparing phase.

Nondeterministic events are certainly a challenge for our approach, because they not only are hard to predict, but also may affect PatternMiner's accuracy to separate static and repeated segments. PatternMiner relies on the developers to abstract away nondeterministic events. In general, PatternMiner will be more effective if there are not many nondeterministic events.

As shown in Section III, PatternMiner predefines a set of regular patterns. Though in our experiments these patterns can cover a majority of arguments, there exist arguments that do not follow these patterns. This can happen for two reasons: first, it is possible that these patterns exist in our trace, but might not follow any of our predefined patterns. This may be improved with more advanced data mining or machine learning techniques. Second, it is also possible that critical information is missing from our trace, so that it is impossible to find the actual pattern. We believe logging can be further improved in two ways: first, we can use compiler techniques to identify critical control flow that affects the corresponding argument in the source code and add logging information there. Second, our approach currently does not analyze messages exchanged among other nodes and they may also provide valuable information.

Also, we use commodity clusters for experiments. It would be interesting to investigate the feasibility of applying our approach to systems with other hardware features [31, 32].

In addition, our work focus on centralized metadata service. In the future, we plan to apply our approach to peer-to-peer services, such as Cassandra [9], a scalable key-value store.

V. Evaluation

Our evaluation tries to answer two questions:

- How well can our approach extrapolate workloads? To be more specific, how much information can PatternMiner predict and how much requires the developer's effort? Furthermore, how accurate is the extrapolated workload compared to the one from a real system?
- Can the extrapolated workload help identify performance problems?

To answer these questions, we have applied our approach to Hadoop to extrapolate workloads for HDFS NameNode and YARN Resource Manager. We use the PUMA MapReduce benchmark [45] to test the NameNode and the HiBench [30] Spark benchmark to test the Resource Manager.

To answer the first question, we report our experience about using PatternMiner. To validate the accuracy of our extrapolated workload, ideally, we should run a real benchmark on a large cluster and compare its workload to the ones extrapolated by PatternMiner. However, the difficulty of gaining a large cluster motivates this work in the first place. As a compromise, we run a medium-scale (500 nodes) experiment on Microsoft Azure and compare its real workloads to the ones we extrapolated from our 12-node experiments.

To answer the second question, we have played the extrapolated workload to NameNode and Resource Manager. Our findings are confirmed either by developer's comments or by our investigation of the source code.

A. Settings

We apply our method to Hadoop 2.7.3. For Spark benchmarks, we run Spark 2.2.0 on YARN. Since both HDFS NameNode and the Resource Manager are built upon Hadoop's RPC library, which already includes the logging functionality, we modify its logging functions as discussed in Section III-A.

We run WordCount, InvertedIndex, TeraSort, and KMeans as the workloads. These benchmarks need to upload input files to HDFS first, by either copying them sequentially to HDFS or copying them in parallel using Hadoop DistCp [28]. Note we must include such data uploading in our extrapolated workload: otherwise, the actual job cannot find input files.

TABLE II
SUMMARY OF TYPES OF NODES IN OUR EXPERIMENTS (* MEANS THERE COULD BE MULTIPLE OF THESE NODES).

Name	Functionality
Application Client	Submit the job
DataNodes (*)	Store data
NameNode	Store metadata
Application Master	Manage a job
Mappers (*)	Execute map tasks
Reducers (*)	Execute reduce tasks
Node Managers (*)	Monitor resource usage
Resource Manager	Assign tasks to nodes

We run all benchmarks with scales ranging from 3 servers to 12 servers as DataNodes and Mappers/Reducers. We use one additional server to run NameNode and Resource Manager. Each server is equipped with an Intel Xeon E3-1231 4-core processor (3.4GHz), 16GB of memory, two hard drives, and 1Gb NIC.

Table II summarizes all types of nodes existed in our experiments. In one experiment, there could be multiple of some types of nodes (denoted with "*"). For example, a TeraSort experiment with 5 DataNodes and 20 GB input data may include 5 DataNodes, 5 Node Managers, 20 Mappers and 10 Reducers.

B. Extrapolating workload

In this section, we report our experience of applying PatternMiner to extrapolate workloads for the NameNode and the Resource Manager.

*1) **Extrapolating for NameNode**:* NameNode maintains file system namespace information and block locations for HDFS. We first present our experience with WordCount for its simplicity and then present additional challenges we met with other benchmarks.

NameNode receives 21 types of RPCs and 150 types of arguments from six types of servers, including DataNode, Mapper, Reducer, Node Manager, Application Master, and Application Client (MapReduce program). During the preprocessing phase, PatternMiner clusters logs into seven types including all six types mentioned above and an additional type of Node Manager, because one Node Manager needs to perform extra tasks.

Patterns of RPC sequences: PatternMiner first identifies nondeterministic RPCs and then separates repeated and static segments.

PatternMiner identifies two types of nondeterministic RPCs: first, when closing a file, a node calls *complete* repeatedly until it returns true, which means the file is fully replicated. Therefore, the number of repetition is nondeterministic, depending on the timing of the *complete* call. We replace such pattern with a single *complete* during analysis, and when testing, we change them back into a *complete* encapsulated in a while loop.

Second, HDFS replicates the job.split file on three DataNodes first and increases its number of replicas later. In this case, the NameNode needs to replicate it on more DataNodes, but the timing of such event is nondeterministic. Interestingly, we find this event is more likely to be a bug: HDFS increases the number of replicas of job.jar and job.split files, which are read by all nodes, to improve their read performance. However, if such increasing happens after the file is read, it is not useful. Actually, we find for job.jar, HDFS creates more replicas when creating the file, which makes the late replication of job.split suspicious. In our analysis, we relocate this event to a deterministic location.

After removing such nondeterministic RPCs, PatternMiner is able to identify static and repeated segments. PatternMiner identifies two types of repeated RPCs in the real-time field: one is the *sendHeartbeat* from each DataNode and the other is the *renewLease* from any server that may open a file. Apart from them, PatternMiner identifies several types of repeated segments in logical-time field in multiple components. For example, the sequential Application Client contains two types. In the first one, the Application Client uses a loop to upload all input files. PatternMiner identifies a sequence of 16 RPCs to upload one file. The second one is related to replicating the job.jar and job.split files. PatternMiner finds each *addBlock* call to these two files is followed by a sequence of *blockReceived* RPCs, whose length increases with the scale of the system. Note for other files, such pattern also exists but since their number of replicas is constant, PatternMiner classifies such pattern as static. Another example is the repeated file renaming segments in Application Master. The RPC sequence of [**getFileInfo, getListing, getFileInfo, rename**] renames each file partition of the input data and scales with the number of partitions.

TABLE III
LENGTHS OF EXTRAPOLATED RPC SEQUENCES FOR NAMENODE UNDER TERASORT (* MEANS THERE COULD BE MULTIPLE OF THESE NODES).

	Data Generation	Sort
Application Client	43	57
DataNodes (*)	4	4
Application Master	82	66
Mappers (*)	40	4
Reducers (*)	0	40
Node Manager A	9	14
Node Managers B (*)	5	10

For example, Table III summarizes the lengths of extrapolated RPC sequences of all types of nodes existed in TeraSort experiments. During a MapReduce job, Application Master and Mappers/Reducers run in containers that are scheduled by the Resource Manager, and managed by the first node manager and other node managers, respectively. And we represent two types of node managers as A and B in the table. To save space, we do not show statistics of other benchmarks and only show the lengths. For example, the representative RPC sequence of DataNodes is [**versionRequest, registerDatanode, blockReport, sendHeartbeat**] and **sendHeartbeat** is repeated in real-time field. In addition, the whole RPC sequence repeats itself across all DataNodes in the experiment.

Patterns of RPC arguments: Next PatternMiner iden-

TABLE IV
MINING RPC ARGUMENTS FOR NAMENODE. C = CONSTANT; R = REGULAR PATTERN (EXCLUDING CONSTANT VALUES); IF = INFORMATION FLOW; U = UNKNOWN

	Total	C	R	IF	U
WordCount	1371	754	47	541	29
InvertedIndex	2011	1130	48	800	33
TeraSort	3134	1710	92	1278	54
KMeans	3178	1773	84	1262	59

tifies patterns for the values of RPC arguments. Table IV summarizes the results of this phase. In Table IV, all numbers are counted after PatternMiner summarizes representative iterations and representative nodes, so that these numbers are consistent across experiments with different scales. The actual total number of argument patterns grows with scale. For example, there are 8053 information flow patterns for a TeraSort experiment of 9 DataNodes and 18 GB input data. To compare these patterns across experiments with different scales, we summarize them to 1278 patterns. For a better understanding, we separate constant values from other regular patterns.

As shown in the table, PatternMiner is able to extract patterns for about 98% of the arguments, among them constant values and information flow from previous RPCs are dominant. Regular pattern mainly involves patterns regularly changed across iteration (e.g. file1, file2, etc) and patterns regularly changed across the same type of nodes (e.g. mapper.0, mapper.1, etc).

We manually investigated the 29 arguments that are marked as "unknown". They can be mainly categorized into a few types: the first type is related to random values (e.g. uuid) or timestamps (e.g. contextid). They can easily be addressed by writing a few lines of code to call a random generator or read the current time. The second type is related to the size of the output file and the local storage usage of a DataNode. We cannot know their accurate values without actual data processing, but since NameNode's performance is not sensitive to the actual value, we use estimated numbers.

InvertedIndex, TeraSort, and KMeans: When applying PatternMiner to more complicated benchmarks, we observe a few additional nondeterministic events:

- In the reduce phase, PatternMiner finds that *blockReceived*s for different *addBlocks* can be merged. For example, if two reducers call *addBlock* simultaneously, the corresponding DataNode may call *blockReceived* only once, but reports both blocks. To handle them during analysis, we separate a merged *blockReceived* into separate ones. To simulate this phenomenon during testing, we need to define a probability to merge *blockReceived*s. Note this phenomenon does not happen in WordCount because it has only one reducer.
- Because of load imbalance among reducers, they may write different amount of data to HDFS, leading to different number of *addBlock*s. Such skew has been thoroughly studied in previous work [34]. We create a parameter to allow a developer to specify the load distribution so that PatternMiner uses it to determine the reducers' load.
- TeraSort needs to sample data in input files for load balance. Instead of listing and handling all input files in one loop, TeraSort lists and handles a certain amount of input files at a time till all input files are handled. We replace them with a single loop in analysis and break the loop into nested loops in actual testing.

TABLE V
LENGTHS OF EXTRAPOLATED RPC SEQUENCES FOR RESOURCE MANAGER UNDER WORDCOUNT (* MEANS THERE COULD BE MULTIPLE OF THESE NODES).

	WordCount
Application Client	5
Application Master	4
Node Managers (*)	2

Table IV summarizes the number of arguments PatternMiner can predict for these benchmarks. The percentage of unknown patterns is close to that of WordCount.

*2) **Extrapolating for Resource Manager**:* The Resource Manager keeps track of resource usage in the system and allocates resource to different jobs. We use both MapReduce and Spark benchmarks to test Resource Manager and focus on Spark benchmarks because they cover more types of RPCs and arguments. In our experiments, it processes 10 types of RPCs and 99 types of arguments from three types of nodes, including the Application Master, the Node Managers, and the Application Client (Spark driver). Since the patterns in four benchmarks are not much different, we only present the result for WordCount.

Patterns of RPC sequences: PatternMiner does not find any nondeterministic RPCs. PatternMiner identifies three types of repeated RPCs in the real-time field: one is the *nodeHeartbeat* RPC from each Node Manager, another is the *getApplicationReport* RPC from Application Client, and the other is the *allocate* RPC from the Application Master. PatternMiner does not find any repeated pattern in the logical-time field.

Table V summarizes the lengths of extrapolated RPC sequences of all types of nodes in WordCount experiments. For example, Application Client has a sequence of 5 RPCs: [**getClusterNodes, getClusterMetrics, getNewApplication, submitApplication, getApplicationReport**] and **getApplicationReport** is repeated in real-time field.

Patterns for RPC arguments: Table VI summarizes the results of this phase. As shown in the table, PatternMiner is able to extract patterns for 83% of the arguments. We manually investigated the remaining arguments that are marked as "unknown". The first type is related to the port, which does not follow any pattern. For them, we just fill a valid port number. The second type is related to the size and creation time of job.jar and job-split.jar: since they vary across experiments, PatternMiner cannot predict them. This problem can be addressed by using a few lines to read file size and timestamp. The third type is the arguments of the periodical *allocate* call: in most cases, the Application Master simply

TABLE VI
Mining RPC arguments for Resource Manager. C = Constant; R = Regular Pattern (excluding constant values); IF = Information Flow; U = Unknown.

	Total	C	R	IF	U
WordCount	179	108	18	22	31

reports its own states and the arguments of these calls are predictable. However, the Application Master also uses this call to ask for new resources when necessary. To predict such information, we write code to simulate the internal logic of the Application Master. The fourth type is the arguments when a node manager reports its task progress: without actually running the task, we have to put estimated numbers.

3) ***Validation and summary**:* Ideally we should run the actual system at a large scale and compare its traces with the ones we extrapolate. Unfortunately, we have not yet been able to gain access to such plentiful resources: after all, it is the very reason that has motivated our work in the first place. The largest validation we have performed involves running a full system with WordCount and TeraSort on 500 nodes in Microsoft Azure [38]. We record their traces to NameNode and Resource Manager. Then, we run small scale experiments of 3, 6, and 12 nodes and use PatternMiner to mine patterns and extrapolate workloads at a large scale of 500 nodes. Finally, we compare real traces of NameNode and Resource Manager to extrapolated workloads. During this procedure, we compare the sequence of RPC function names, their argument values, and the timing between two consecutive PRCs.

For the sequence of RPC function names, we find the traces of the Resource Manager match with our extrapolated workloads. And most of the traces of the NameNode match with our extrapolated workloads with one exception in the data generation phase of TeraSort: 3 DataNodes in Azure failed during our experiments, leading to different number of *blockReceived* for about 3.4% of the blocks within a few mappers. In our small-scale experiments, DataNodes did not fail and thus we cannot predict them. As discussed in Section IV, to improve accuracy, one could trigger failures deliberately in small-scale experiments.

For RPC arguments, both traces of the NameNode and Resource Manager match with extrapolated workloads. Note that our extrapolated workloads may provide rules instead of actual values for some arguments (e.g. information flow from previous RPCs): in these cases, we check whether the actual values in real traces match with these rules.

For time interval between two consecutive RPCs, the difference between real and extrapolated traces is within 10% for 90% and 99% of the intervals for NameNode and Resource Manager, respectively. Intervals with large difference mainly come from a few nodes when the system experiences burst traffic. In addition, the start time of a node for both traces match with extrapolated workloads.

To summarize, PatternMiner requires developers' effort to handle nondeterminstic events and unknown argument patterns. For NameNode, our experience is that handling and simulating merged *blockReceived* requires a reasonable amount of effort and all others are easy to handle. For Resource Manager, our experience is that predicting when the Application Master will ask for new resource through *alloc* is reasonably challenging, and all others are easy to handle.

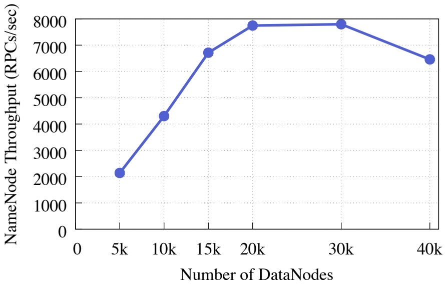

Fig. 5. Scalability of NameNode under TeraSort (We collocate 5,000 nodes on each machine with 1 container on each Node Manager).

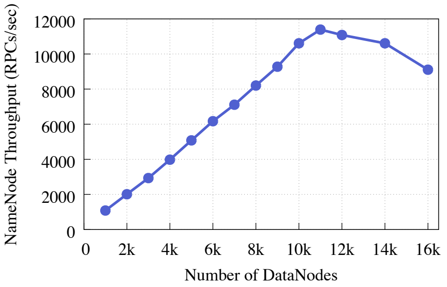

Fig. 6. Scalability of NameNode under TeraSort (We collocate 1,400 nodes on each machine with 8 containers on each Node Manager).

C. Evaluating at scale

We play the extrapolated workload to NameNode and Resource Manager to measure their capacity to know the maximum scale of the system, and identify potential performance bottlenecks.

NameNode: For all four benchmarks, we are able to run the extrapolated workloads successfully with different scales and we are even able to run a mixture of them. Given the complex dependencies among RPCs, such success indirectly validates the accuracy of our approach. We show results from TeraSort because it incurs the heaviest load on NameNode. The degree of collocation we can achieve depends on the number of containers we put on each Node Manager. If we put one container on each Node Manager, then we can collocate 5,000 nodes (each node runs one DataNode process, one Node Manager process and and one Mapper/Reducer process) on one physical machine: the largest experiment we run use a workload generated for 40,000 nodes and we are able to run it on only 8 physical machines. If we put eight, then we can collocate 1,400 nodes (each node runs one DataNode process, one Node Manager and 8 Mapper/Reducer processes) on one physical machine.

As shown in Figure 5, if we put one container on each Node Manager, NameNode can support up to about 20k nodes

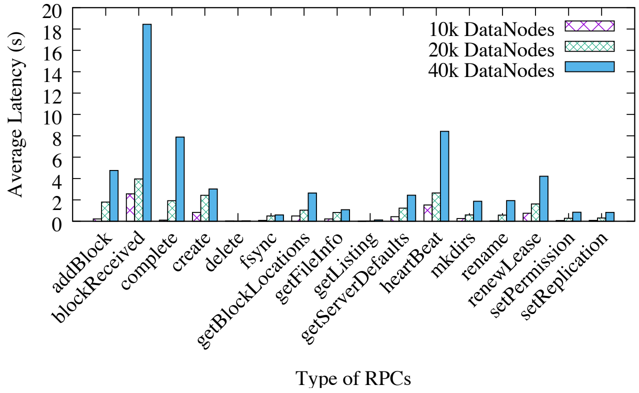

Fig. 7. Latency of each type of RPC with 1 container on each Node Manager (three types are omitted since they are only called during initialization).

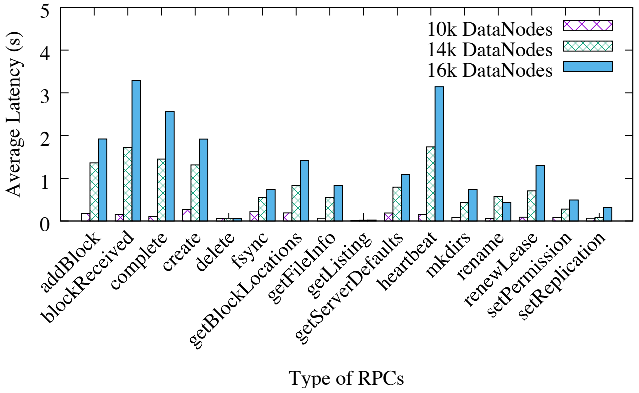

Fig. 8. Latency of each type of RPC with 8 containers on each Node Manager (three types are omitted since they are only called during initialization).

under the TeraSort benchmark: afterwards, the NameNode is overloaded. When running these experiments, we find one correctness issue: occasionally NameNode will report a DataNode as failed because the NameNode does not receive the DataNode's heartbeat in time. This may cause the TeraSort job to fail because TeraSort's intermediate data is only one-way replicated and thus any DataNode failure can block the whole job. Our analysis of the log shows that the cause of this problem is the burst of DataNodes' heartbeat messages together with the burst of Mappers' or Reducer' RPCs. We alleviated this problem by randomizing the startup of DataNodes so that they won't send heartbeats at the same time. Although this trick allows us to run our system to a larger scale, it is not a fundamental solution and such problem could happen under a heavier load.

This problem demonstrates the necessity of our work: certain problems only happen at a large scale, when a node is under heavy load. Furthermore, such problems not only hurt performance, but may also cause severe consequences like failing a large job. The recent development of newer versions of HDFS confirms our finding: HDFS 2.8 adds a feature called DataNode Lifeline Protocol to "prevent the NameNode from incorrectly marking DataNodes as stale or dead in highly overloaded clusters where heartbeat processing is suffering delays" [19].

Figure 6 shows the result for the case of Node Manager with 8 containers. NameNode throughput grows linearly with scale and achieves peak performance with about 11k DataNodes under the TeraSort benchmark: afterwards, the NameNode is overloaded. As shown in Figure 5 and 6, the scalability of the NameNode depends on the system configuration. Putting more containers on one NodeManager will reduce the number of nodes the NameNode can support, but when considering the total number of containers, it actually improves the scalability of HDFS.

In addition, Figure 7 and 8 shows the average latency of each type of RPC with 1 and 8 containers on each Node Manager: $blockReceived$, $complete$, and $heartBeat$ are on top. We investigate the source code of NameNode and find that they are all heavily contending operations: $heartBeat$ needs to hold a lock for almost all of its execution, leading to long latencies under heavy load and causing the problem mentioned above. $blockReceived$ and $complete$ need to grab the same write lock during their execution, so they cannot execute in parallel. It may be possible to reduce such contention by using fine-grained locking, but it is out of the scope of this paper because it requires significant changes to NameNode.

Resource Manager: Traditional metrics like throughput and latency do not work well with the Resource Manager (RM): when an application asks for containers from the RM by calling *allocate*, the RM may just return 0 if it does not have resource and the application will retry later. The overhead of returning 0 is certainly much smaller than actually allocating the resource. To accurately measure the performance of the RM, we define a new metric called startup latency, which is the latency from the application registering to the RM to the application getting all containers from the RM.

We start enough tasks, each requesting 200 containers, to consume all container resources in the cluster. As shown in Figure 9, the startup latency steadily grows with scale and increases sharply when the scale reaches 60,000 containers, which means the Resource Manager is saturated there. Even for medium scales, ten seconds of delay to start the application will be problematic for short tasks.

In addition, we evaluate the impact of the number of containers on each Node Manager. We start 20 jobs, each requesting 200 containers. We also fix total number of containers to 4,000, configure the number of containers and adjust the number of Node Managers accordingly. Figure 10 shows that larger number of configured containers incurs longer startup latency.

Besides, we have observed an over-subscription problem: an application may get more containers than it asks for. Our investigation shows this is caused by a race condition: in YARN, when an application asks for a number of containers, the RM may give containers in multiple batches; and if the application does not get all it needs, it can ask for the remaining ones. In this model, if it happens that the RM

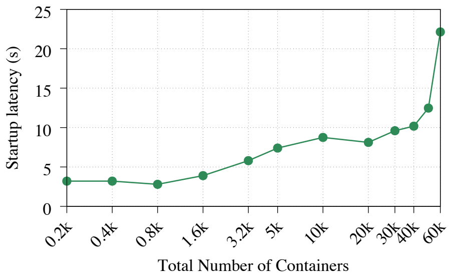

Fig. 9. Average startup latency when testing the Resource Manager with different total number of containers

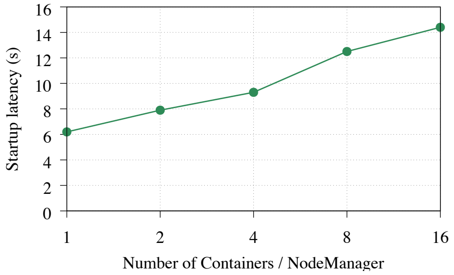

Fig. 10. Average startup latency when testing the Resource Manager with different number of containers on each Node Manager

gives the last batch and the application asks for the last batch at the same time, the RM may consider the request as a new one and thus will give more than necessary. This issue is confirmed in the community: we find online discussions about this problem [16]; Spark has a fix for this problem to give back unnecessary containers; we don't find similar fix in MapReduce though. Our work can simplify the debugging and investigation of this issue because such multi-batch allocation is more likely to happen in larger tasks.

VI. Related work

Evaluating system at large scale: Large-companies may have the resource to test their system at large scale. For example, Yahoo has conducted a 4000-node experiment on HDFS [52]. Facebook's Kraken [57] system can intentionally redirect some live traffic to stress test its web servers. However, on the one hand, it is hard for most researchers in academia to access such resource. On the other hand, using a production system for testing incurs the risk of unexpected failures or slowdown.

For researchers who do not have such abundant resource, two common practices—resource extrapolation and synthetic workload—are widely used. In previous works, resource extrapolation is used, among others, in RAMCloud [44], Spanner [17], and Salus [59], while synthetic workload is used to evaluate HDFS [52, 58] and HBase [58]. Since Section I has already discussed these two approaches, we do not discuss them further here.

David [2] and Exalt [58] automatically synthesize workloads for storage servers. David [2] leverages the observation that the performance of a local file system does not depend on the contents of files. Thus, David only stores the file system's metadata and discards its data. Exalt [58] extends this idea to distributed storage systems by separating data and metadata in a multi-layer distributed system. However, for systems that rely on content of data to function, these approach do not work. For example, Exalt cannot run MapReduce jobs because these jobs need to know the contents of data.

Scale Check [35] extends the intuition of Exalt. It replaces CPU-intensive processing with sleep() in order to collocate a large number of nodes on one machine and help developers find and replay scalability bugs with high accuracy. However, how to automatically find safe places to perform such replacement turns out to be challenging.

Dynamometer [22] is a tool that targets evaluating the Hadoop's HDFS NameNode. It also keeps track of metadata, and creates a simulated cluster of one NameNode and many DataNodes to replay production workload with a few machines. However, it is a specific tool for NameNode and cannot scale beyond the production workload.

DieCast [26] uses the idea of timing dilation [27]: it runs multiple servers inside virtual machines on a single physical machine. It slows down each data server by a constant factor, measures system throughput, and finally multiples the measured throughput by the same factor. DieCast can collocate a certain number of data servers on a single physical machine when CPU is the bottleneck, but does nothing to reduce the storage requirement for systems that are I/O heavy.

Workload extrapolation: A number of works analyze the workloads in the past to predict the workload in the future, by using techniques like Markov model [43] and time series modeling [54]. Our work tries to use workloads from small-scale experiments to predict the workload at a larger scale.

Log analysis: Log analysis has been used extensively in previous works to infer causal relationship between events [14, 36, 65], to understand system behavior [6, 23, 25, 62, 66], and to debug correctness and performance problems [12, 41, 49, 60, 63]. Our work relies on part of these techniques (e.g. causal relationship), but it has a completely different goal: our work tries to extrapolate how workloads change with the scale of the system.

VII. Conclusion

Testing a scalability bottleneck is challenging because it requires a large number of nodes. This paper shows that we can test a bottleneck node by extrapolating a workload that the node would observe at a large scale.

Our case studies have confirmed the feasibility and effectiveness of workload extrapolation. We plan to further improve its applicability and accuracy by incorporating compiler and advanced data mining techniques.

Acknowledgements

Many thanks to the anonymous reviewers for their insightful comments. This material is based in part upon work supported by NSF grant CCF-1747447.

References

[1] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA.

[2] Nitin Agrawal, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. In *FAST*, 2011.

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, 2017.

[4] Amazon EC2. http://aws.amazon.com/ec2/.

[5] Apache HBASE. http://hbase.apache.org/.

[6] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[8] Canada's immigration website crashes during US vote. http://www.bbc.com/news/technology-37921376, 2016.

[9] Cassandra. http://cassandra.apache.org.

[10] Chameleon cloud. https://www.chameleoncloud.org/.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[12] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*.

[13] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, 2011.

[14] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[15] Cloudlab. https://www.cloudlab.us/.

[16] Allocation of excess containers. https://issues.apache.org/jira/browse/YARN-1902.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[18] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[19] DataNode Lifeline Protocol. https://issues.apache.org/jira/browse/HDFS-9239.

[20] Jeff Dean. The Rise of Cloud Computing Systems. http://sigops.org/sosp/sosp15/history/10-dean-slides.pdf, 2015.

[21] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. Querying and mining of time series data: Experimental comparison of representations and distance measures. *Proc. VLDB Endow.*, 1(2):1542–1552, August 2008.

[22] Dynamometer. https://github.com/linkedin/dynamometer.

[23] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, 2007.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[25] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC, Berkeley, CA, USA, 2011.

[26] Diwaker Gupta, Kashi Venkatesh Vishwanath, and Amin Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *NSDI*, 2008.

[27] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *NSDI*, 2006.

[28] Hadoop DistCp Tool. https://hadoop.apache.org/docs/current3/hadoop-distcp/DistCp.html.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[30] Intel HiBench. https://github.com/intel-hadoop/HiBench.

[31] J. Zhang, X. Lu, J. Jose, M. Li, R. Shi, D. K. Panda. High Performance MPI Library over SR-IOV Enabled InfiniBand Clusters. In *Proceedings of International Conference on High Performance Computing (HiPC)*, Goa, India, 2014.

[32] J. Zhang, X. Lu, J. Jose, R. Shi, D. K. Panda. Can Inter-VM Shmem Benefit MPI Applications on SR-IOV based Virtualized InfiniBand Clusters? In *Proceedings of 20th International Conference Euro-Par 2014 Parallel Processing*, Porto, Portugal, 2014.

[33] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC*, 2008.

[34] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 2012.

[35] Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability bugs: When 100-node testing is not enough. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, 2017.

[36] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[37] Macy's Web Site Buckles Under Heavy Traffic on Black Friday. http://fortune.com/2016/11/25/macys-black-traffic/, 2016.

[38] Windows Azure Platform. http://www.microsoft.com/windowsazure/windowsazure.

[39] Characterization of the DOE Mini-apps. https://portal.nersc.gov/project/CAL/doe-miniapps.htm.

[40] David W. Mount. *Bioinformatics: sequence and genome analysis*. Cold Spring Harbor Laboratory Press, 2004.

[41] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[42] NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html.

[43] James Oly and Daniel A. Reed. Markov model prediction of i/o requests for scientific applications. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, 2002.

[44] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[45] PUMA Benchmarks. https://engineering.purdue.edu/~puma/pumabenchmarks.htm.

[46] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, D. K. Panda. Designing efficient small message transfer mechanism for inter-node mpi communication on infiniband gpu clusters. In *2014 21st International Conference on High Performance Computing (HiPC)*, Goa, India, 2014.

[47] R. Shi, S. Potluri, K. Hamidouche, X. Lu, K. Tomko, D. K. Panda. A scalable and portable approach to accelerate hybrid HPL on heterogeneous CPU-GPU clusters. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Indianapolis, IN, USA, 2013.

[48] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, D. K. Panda. Hand: A hybrid approach to accelerate non-contiguous data movement using mpi datatypes on gpu clusters. In *2014 43rd International Conference on Parallel Processing*, Minneapolis, MN, USA, 2014.

[49] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, 2006.

[50] Paul Saab. Scaling memcached at Facebook. https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919/, 2008.

[51] Rong Shi and Yang Wang. Cheap and available state machine replication. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, Berkeley, CA, USA, 2016.

[52] Konstantin Shvachko. HDFS scalability: the limits to growth. http://c59951.r51.cf2.rackcdn.com/5424-1908-shvachko.pdf.

[53] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[54] Nancy Tran and Daniel A. Reed. Automatic arima time series modeling for adaptive i/o prefetching. *IEEE Trans. Parallel Distrib. Syst.*, 15(4):362–377, April 2004.

[55] Transaction Processing Performance Council. The TPC-W home page. http://www.tpc.org/tpcw.

[56] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[57] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[58] Yang Wang, Manos Kapritsos, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *NSDI*, 2014.

[59] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus scalable block store. In *NSDI*, 2013.

[60] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[61] Yahoo! Computing Systems Data. https://webscope.sandbox.yahoo.com/catalog.php?datatype=s.

[62] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[63] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[64] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6), February 2017.

[65] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 603–618, GA, 2016. USENIX Association.

[66] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.