Supplemental Material: Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures

Ting Dai, Member, IEEE, Daniel Dean, Member, IEEE, Peipei Wang, Member, IEEE, Xiaohui Gu, Senior Member, IEEE, and Shan Lu, Senior Member, IEEE

1 ADDITIONAL RESULTS

In this section, we discuss four additional cases to discuss Hytrace’s bug inference results in detail.

1.1 Additional Case Study

Apache-45856 (C/C++): In this bug, when suexec_log is larger than 2GB, corresponding CGI and SSI applications, which are using the suEXEC feature of Apache server, hang. The root cause of this hang is in function err_output, which uses fopen to open and append large files (larger than 2GB) on a 32-bit machine.

Hytrace can identify the root cause function err_output and have improved this function’s ranking significantly from the PerfScope result (41th up to 1st). Specifically, when the performance anomaly happens, Hytrace-dynamic identifies both function err_output and its caller log_err, which invoke system calls with abnormal frequencies. Hytrace rule checker has kept the root-cause function err_output in its result, because this function matches “constant parameter”, “unsafe function”, and “uncovered branch” rules. With some functions, which were originally higher ranked by Hytrace-dynamic, not matching any Hytrace rules, the ranks of err_output and its caller gets improved a lot. Clearly, the “unsafe function” rule matched with err_output is exactly the root cause.

Cassandra-5064 (Java): Users reported that sometimes Cassandra would hang as soon as an ALTER TABLE request is issued. The hang actually happens in a while loop in reload function, as shown in Figure 1. In this loop, maybeSwitchMemtable processes every memtable in a list (line 174), until there is no remaining memtable in the list (line 172–173). Clearly, maybeSwitchMemtable should remove a memtable mt from the list after mt is processed. Unfortunately, this is only done for dirty memtables (line 650–652), but not clean memtables. As a result, the while loop in reload becomes infinite, where maybeSwitchMemtable keeps getting invoked to process the same clean memtable again and again, endlessly.

Hytrace identified both maybeSwitchMemtable and reload as rank two suspicious functions. Specifically, Hytrace-dynamic detected maybeSwitchMemtable because certain system calls are invoked much more frequently when the bug is triggered. Hytrace-dynamic then adds reload to the suspicious function list, because it is the caller of maybeSwitchMemtable. Hytrace rule checker did not prune out these two functions, as they both match the “constant parameter” rule, and reload also matches the “unchanged loop exit condition variables” rule.

The “constant parameter” rule matched with reload is the direct cause, while the “unchanged loop exit condition variables” rule matched with reload is related to the root cause of the observed performance problem. Specifically, reload invokes maybeSwitchMemtable with a constant parameter, True (line 174). As a result of this constant True, expensiveCommitLog.discardCompletedSegments function is always invoked inside maybeSwitchMemtable (line 694).
Fig. 2. Partial call graph for Tomcat-53173 bug.

```java
public void run()
{
    ...  
    986    countUpOrAwaitConnection();
    ...  
}

protected void countUpOrAwaitConnection()
{
    ...  
    671    LimitLatch latch = connectionLimitLatch;
+    if (latch == null && latch.limit == -1) return;
    672    if (latch != null) latch.countUpOrAwait();
}
```

Fig. 3. Partial call graph for Mapreduce-3738 bug.

```java
public void run()
{
    ...  
    193    this.appAggregationFinished.set(true);
    ...  
}
```

And all of the above operations keep happening in the `while` loop (line 168) without updating any loop exit condition variables, consuming a lot of CPU and disk resources and causing the performance problem observed by users.

**Tomcat-53173 (Java):** Users reported that sometimes Tomcat would hang as soon as `maxConnections` is set to be `-1`. The hang happens because Tomcat is stuck inside the `countUpOrAwaitConnection` function, as shown in Figure 2 (the value of `maxConnections` is passed to `latch.limit` and `limit`). When Acceptor thread processes incoming connections, it calls function `countUpOrAwaitConnection` (line 986). In theory, setting `maxConnections` as `-1` means putting no upper-limit to accepting client socket connections. Consequently, `countUpOrAwaitConnection` should return immediately without any waiting. Unfortunately, this special setting (i.e., `-1`) is not specially handled. Instead, function `latch.countUpOrAwait` is invoked to try fetching a lock. This lock fetching will never succeed, as indicated by line 39 and line 41 in function `tryAcquireShared` — when `limit` is `-1`, the line-39 condition is always true and hence the function always returns `-1`, indicating a lock-acquisition failure. The execution then gets stuck in repeatedly trying to acquire the lock, while the client’s connections get blocked.

Hytrace identified `countUpOrAwaitConnection` as a rank 10 suspicious function. Specifically, Hytrace-dynamic detected `countUpOrAwaitConnection` because it invokes a set of system calls with abnormal frequencies in performance-anomaly runs. Hytrace rule checker did not prune out this function, as it matches with the “uncovered branch” rule — line 672 in Figure 2.

The uncovered-branch rule matched with `countUpOrAwaitConnection` is related to the root cause of the observed performance problem. The patch exactly added more handling for more branch scenarios around line 672, as shown in Figure 2.

**Mapreduce-3738 (Java):** In our previous paper [1], Hytrace failed to diagnose the Mapreduce-3738 bug. We have re-done the experiments and found that the miss detection is caused by missing the profiles for the root cause functions. After adding the missing profiles back, Hytrace successfully identifies the root cause function `AppLogAggregatorImpl.join` and ranked it the 4th.

We now describe this bug in detail. As shown by Figure 3 once an uncaught runtime exception (e.g., `OutOfMemoryError`) happens in the function `AppLogAggregatorImpl.run`, the true-setting for a variable `appAggregationFinished` could be skipped (line 193). NodeManager will then hang during shutdown by calling `AppLogAggregatorImpl.join`, waiting for `appAggregationFinished` to become true forever (line 253). The patch simply moves the `set(true)` into a `finally` block, which guarantees the execution of `set(true)` even when an uncaught exception happens.

Hytrace identifies `AppLogAggregatorImpl.join` as a suspicious function and have improved this function’s rank from the PerfScope result (12th to 4th). Specifically, Hytrace-dynamic detected `AppLogAggregatorImpl.join` because it invokes system call sequence \{`sys_futex`, `sys_stat64`, `sys_stat64`, `sys_futex`\} with abnormal frequencies. Hytrace rule checker did not prune out this function, as it matches with the “unchanged loop exit condition variables” rule (line 253). In addition, Hytrace rule checker also identifies `AppLogAggregatorImpl.run` as a suspect function, as the invocation of `set(true)` matches the “constant parameter” rule. However, the dynamic component of Hytrace fails to identify `run` function. The reason is that Hytrace-dynamic looks for abnormal system-call related runtime behavior changes. `AppLogAggregatorImpl.run` itself and its callee functions do not issue many system calls and hence are not identified as abnormal.

**REFERENCES**