Model-Based Fault Localization in Large-Scale Computing Systems

Naoya Maruyama[†] and Satoshi Matsuoka[†,‡]
[†] Tokyo Institute of Technology
[‡] National Institute of Informatics
naoya.maruyama,matsu@is.titech.ac.jp

Abstract

We describe a new fault localization technique for software bugs in large-scale computing systems. Our technique always collects per-process function call traces of a target system, and derives a concise execution model that reflects its normal function calling behaviors using the traces. To find the cause of a failure, we compare the derived model with the traces collected when the system failed, and compute a suspect score *that quantifies how likely a particular part of call traces explains the failure. The execution model consists of a call probability of each function in the system that we estimate using the normal traces. Functions with low probabilities in the model give high anomaly scores when called upon a failure. Frequently-called functions in the model also give high scores when not called. Finally, we report the function call sequences ranked with the suspect scores to the human analyst, narrowing further manual localization down to a small part of the overall system. We have applied our proposed method to fault localization of a known non-deterministic bug in a distributed parallel job manager. Experimental results on a three-site, 78-node distributed environment demonstrate that our method quickly locates an anomalous event that is highly correlated with the bug, indicating the effectiveness of our approach.*

1 Introduction

Root causes of software faults in parallel and distributed computing environments, such as HPC clusters and grids, are notoriously hard to localize due to their scale and heterogeneity. Ever increasing scale of current HPC clusters can make even a simple bug take days to find its root cause. Heterogeneous configurations of user environments can cause hard-to-reproduce, rare faults, where such standard engineering disciplines as in-house pre-release testing would be less effective.

Several trace-based anomaly detection techniques have been proposed to semi-automate the fault localization process in such environments [4, 7, 12, 14, 16]. Most of them employ a centralized approach, assuming that traces distributed over remote machines can be collected to a centralized trace analyzer without scalability limit. Although such a centralized approach could allow one to comprehend higher-level control and data flows among distributed nodes [4, 7, 12, 14, 16], their applicability to the current HPC clusters, where hundreds to thousands of machines are not exceptional but commonplace, is still unclear.

To aid human analysts in localizing faults in large-scale computing environments, we propose an automated model-based fault localization method that views the localization as an anomaly detection problem. Our method uses function call traces, which have been shown to be effective in fine-grained fault localization [15]. Given fault traces, our goal is to locate specific function call sequences that are highly correlated with the given fault. To do so, our method consists of two phases: pre-fault model derivation and anomaly detection in the fault traces using the derived model. The first phase, using traces collected under normal operations, automatically derives an *execution model* that reflects the normal function calling behaviors of the target system. When a fault actually happens in the system, the second phase locates specific calling sequences in the traces that are highly correlated with the fault.

The key challenge in realizing such model-based fault localization is how to learn an accurate execution model for fault localization in an automated, scalable manner. Learning accurate models requires to identify self-similar, repetitive behaviors in target systems. While we assume that they exhibit historical similarity, automatically identifying similar behaviors in raw function call traces is not a trivial problem. Often raw traces are very huge in size since typical processes in our target domain do not start and stop in a frequent manner as in desktop applications, but rather always run with little downtime. Furthermore, as stated above, such model derivation must work at scale. For example, the InTrigger distributed computing platform consists of six clusters distributed among Japanese universities and research laboratories. Decentralization and scalability are key factors in analyzing faults occurring in such distributed environments.

To tackle the above challenge, we exploit two observations on typical software architectures and fault characteristics in distributed systems. First, we see that many of the distributed software for clusters and Grids, such as batch job schedulers and parallel file systems, consist of processes that employ the event-driven architecture, where several different event-processing routines are multiplexed into a single event loop. For example, a batch job scheduler for clusters could employ a daemon process on each node whose responsibilities include monitoring of jobs under the node and handling of requests from the master job scheduler. A typical software architecture for such purposes would model job status changes and incoming requests from the master as events, and consist of an infinite loop of event-processing routines. Second, we have observed with our previous work [15] and results by other researchers [16] that many anomalies manifesting themselves over distributed nodes also exhibit locally-observable deviant behaviors. For instance, a bug discussed in [16] caused an event handler function not to be called, failing to serve incoming requests; such a behavior would be detectable using locally-observable information, namely function calling behaviors in this particular example.

Based on the above observations, our approach models system executions by first learning per-process function calling behaviors, and then aggregating them into a single model that represents the behaviors of the entire system. Specifically, for each member process, we generate a concise per-process model called *process model* from its function call traces. Unlike the previous approaches that attempt to reconstruct distributed flows by matching distributed traces [4, 7, 12, 14, 16], we only use the local information to generate the process model in order to eliminate centralized bottlenecks.

To derive the process model, we first decompose the entire function traces into sub traces, or *execution units*, based on its associated event source, and then derive a model for each event source. As a type of events, this paper focuses on network events, which we believe would be the most important events in distributed computing systems. The event source of network events is its connection; thus, we treat a sequence of function calls corresponding to the same network connection as a single execution unit. Next, for each connection, we derive a model by constructing a call tree of every function appearing in its associated units, and assigning each function an estimated probability of appearance. We estimate the probability of a function by dividing the number of its occurrences by the total number of occurrences of the execution units for the same connection. For instance, if a function always appears when any message arrives at a connection, we give the function probability 1. Creating separate models for different connections improves the accuracy of the probability estimation, since different connections are likely to have different function calling behaviors. Finally, we derive the *global model* by merging the process models whose processes are inferred to have played the same role in the system.

We aid human analysts in localizing the root cause of a fault by comparing its fault traces with the derived model. Given the fault traces, we first decompose them into execution units as in the model derivation. For each execution unit, we compute a *suspect score* that quantifies how likely a particular part of call traces are correlated with the fault. Functions with low probabilities in the model give high suspect scores when called upon a failure. Frequently-called functions in the model also give high scores when not called. Finally, we report the execution units ranked with the suspect scores to the human analyst, narrowing further manual localization down to a small part of the overall system.

Both our modeling and fault localization operate in a mostly-decentralized fashion. In the model derivation phase, only the derivation of global models requires a globally-coordinated centralized operation, while the derivation of process models analyzes raw traces in parallel using the same set of nodes as the target sytem. Once the global model is derived and deployed to each local node, our fault localization requires no remote operations. Therefore, our method can achieve higher scalability compared to the previous approaches based on centralized algorithms [4, 7, 12, 14, 16].

We have applied our proposed method to localizing a known non-deterministic bug in a distributed job manager. Experimental results on a three-site 78-node distributed environment demonstrate that our method quickly locates an anomalous event that is highly correlated with the bug. Specifically, without our automated trace analysis, we would have needed to examine all traces of 78 nodes accounting for a complete 70-second run. Our localization analysis narrowed the localization only to the traces from two nodes for less than a second, significantly reducing the fault localization burden.

2 Model Derivation

logic anomalies. A logic anomaly is a situation where an intended operation is not performed or a non-intended operation is performed. Such misbehavior often leads to different function coverage. For instance, a bug discussed in [16] caused an event handler function not to be called. Another example is a bug that caused a hang in a distributed job manager presented in Section 4.

The bug caused a function that had never been called in normal operations to be called.

We derive the execution model so that it can give quantitative differences between traces with such logic anomalies and normal traces. To do so, we consider the following two classes of functions as having higher suspicions:

- Functions that are rarely called in normal operations, but are called when a fault happens.
- Functions that are often called in normal operations, but are not called when a fault happens.

Our model quantifies how such properties hold between normal and fault traces with the estimated function call probabilities. We compute the probabilities by obtaining known-normal function call traces from the same system under normal operation states. Using the collected normal traces, we derive the execution model via the following three-step process: 1) decomposition to execution units, 2) process model derivation, and 3) global model derivation. The result of this process is a global model consisting of multiple process models, each of which reflects the normal function call probabilities of a particular process group. The rest of this section describes the details of each step.

2.1 Function Call Tracing

We use per-process function call traces for learning execution models for the following three reasons. First, it gives fine-grained localization resolution compared to other approaches that use higher-level system properties, such as nodes, processes, and software components [4, 7, 12]. Second, it is relatively easy to obtain compared to finer-grained information such as branch profiles. Modern programming languages, especially VM-based ones such as Java and Python, often provide a built-in function tracing capability. Even a popular C compiler gcc provides a compile-time function call hook framework. Furthermore, use of a binary instrumentor such as Dyninst [6] can produce function traces of binary programs as well. Third, function traces can often be taken with acceptable performance overhead. In our experimental studies presented in Section 4, we have observed less than 7% of slowdown of application performance running under a traced job manager.

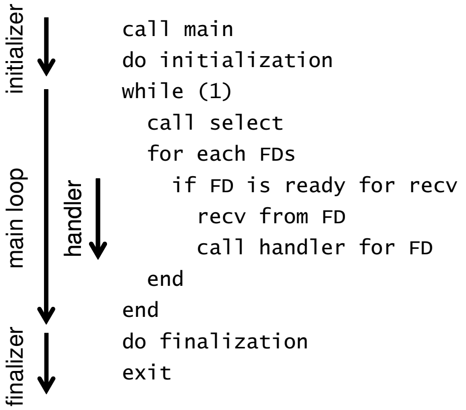
A trace entry consists of four fields—type, timestamp, caller, and callee—and a variable number of optional fields. The type field consumes one byte and designates the type of the entry; we have currently three types: call, return, and exception. For ease of the modeling and anomaly detection, we also encode unique identifiers for particular functions such as `connect` and `recv`. The timestamp field records the value of the CPU cycle counter, and consumes seven bytes. The caller and callee fields record the address of the caller and callee, respectively. The size of the fields depends on architectures: two bytes and six bytes on the x86 architecture and its 64-bit extension x86-64, respectively. Note that while the x86-64 architecture allows 64-bit addressing, the current available products do not use the top-most two bytes [1, 11]. We omit those two bytes for reducing the trace size. The optional fields encode the parameter and return values for particular functions. For example, for the trace decomposition described in Section 2.2, we record the value of the file descriptor parameter of the `recv` system call. The largest optional field in our current implementation is 128 bytes for recording the ready file descriptor numbers on returns from the `select` system call. Overall, the total size of a trace entry ranges from 16 bytes to 148 bytes.

2.2 Decomposing Traces into Execution Units

Once function traces are obtained, we decompose them into sub traces, or *execution units*, based on their associated events, assuming that the target system employs an event-driven architecture. By separating event-handling routines into different execution units, we aim to improve the resulting accuracy of the derived model. Among various events in distributed systems, this paper focuses on network events such as arrivals of incoming messages since they are one of the most representative classes of events in distributed systems. Figure 1 illustrates an example of such trace decomposition. Note that a program can include regions that are not related with any meaningful network events. For example, the function calling behaviors in the program initialization and finalization parts should not depend on any incoming messages. We also identify such parts as different units.

We decompose traces into execution units by the following three-step process of automated trace and program analyses. We describe here an algorithm for programs that use the `select` system call for event multiplexing. It should be easily extended to other event-processing frameworks, such as polling, as well.

Step 1: Identification of the event loop Since we assume that our target system uses `select` to multiplex message-handling routines, we can also assume that the event loop must include calls to `select` and `recv`. Based on this observation, we find the event loop as follows. First, we detect loops at run time by detecting recurring call stacks in the function traces. We consider the call sequence inside the recurring stacks as a loop body trace. Next, among the detected loops, we locate the loop that has calls to `select` and `recv` in its body



```
call main
do initialization
while (1)
  call select
  for each FDs
    if FD is ready for recv
      recv from FD
      call handler for FD
  end
end
do finalization
exit
```

Figure 1. An example of trace decomposition into execution units.

by analyzing the program source or binary code, and determine it as the event loop.

Step 2: Identification of the handler units We decompose the traces inside the loop at every call to `accept` or `recv`, and determine a sequence of trace entries from a call to the next as a *handler unit*. The trace decomposition at `recv` calls assumes that a single event handling starts by receiving a message at first and performs any operations according to the received message content without further receiving data from the connection. Note that a `recv` system call can return only a partial message when the given receive buffer is too small to copy the whole message. Thus, the typical use case of the API repeats a call to `recv` until no more data is available on the connection. We treat consecutive calls to `recv` with the same file descriptor as a single `recv` call.

Step 3: Identification of the other units The part of traces from the very beginning of the program to the start of the event loop should represent the initialization of the program, which typically includes such operations as listening-port setup and event handler registration. We call the part the *initializer unit*. In addition, the calls after running the event loop until the process exits should represent its finalization; we call the part *finalizer unit*.

2.3 Process Model Derivation

A *process model* consists of call trees of the functions in the decomposed execution units. We derive the process model in the following automated analysis. Note that we perform this model derivation at the same node as each target process, making it completely decentralized.

First, for each unit, we construct a tree representing function calls from the starting function of the unit. Each node n corresponds to a function call annotated with its call site s and callee function g, denoted as $s \to g$. The call site s denotes a unique location in the parent node's function f. The path from the tree root to each child node corresponds to a function call stack executed at its trace collection time. We allocate different nodes to calls from different call sites to the same function, i.e., if $n = s_1 \to g$ and $m = s_2 \to g$, then $n \neq m$. However, we allocate a single node to multiple calls to the same function from the same location, in order to avoid the size of the tree growing excessively due to a large number of loop iterations.

Next, we merge the handler units based on their associated connections so that the process model has a unique sub model for each event source. We define the equality of connections by their call stacks to the connection-establishing functions, including `bind`, `connect`, `listen`, and `accept` of the Berkeley Socket API. Specifically, let h be a handler unit, and c be the connection from which the unit received a message. Let s be the call stack to one of the connection establishing functions for c, denoted as $s = \{n_0, \ldots, n_k\}$, where the stack originates from call tree node n_0 and ends with n_k. Let $S(c)$ be the set of all the call stacks s associated to connection c, i.e., $S(c) = \{s_i\}$. We consider two connections, c_1 and c_2, to be equal if and only if their associated call stacks are the same, i.e., $S(c_1) = S(c_2)$. Based on this equality relation, we categorize handler units into multiple groups where handler units in the same group have the equal connection each other. For each group, we create a single call tree by merging the trees of the member units.

The observation behind the above handler unit merging is that we expect that if two connections are established by the same call stacks, the handler units associated with them should include similar function calls. This is not necessarily the case: for example, if multiple connections are established by a single call site to `connect` in a loop with different socket descriptors, and those connections are actually related to different roles in the program. However, we expect that such a program structure would be rather rare; in fact, our case study presented in Section 4 exhibits no such behavior.

Finally, we annotate each node of the call trees by its estimated call probability. We estimate the probability of a node by counting the number of occurrence of execution units where the call-stack path of the node appears. Let f and g be calls in a tree where f is the parent of g. Also let n_f and n_g be the number of execution units where f and g appear, respectively. We compute the estimated occurrence probability of g as

$p_g = n_g/n_f$. Note that we do not consider how many a node appears in a unit (i.e., frequency), but only whether it appears or not (i.e., coverage). Thus, n_g is always less than or equal to n_f.

2.4 Global Model Derivation

Once process models are created, we gather them to a central location, and merge them into a single *global model* so that it includes the function call behavior of each one of the member processes. The reason to merge process models is the scalability with respect to number of processes. The space cost to keep individual models for, e.g., hundreds of thousands of processes would be prohibitively high. For example, in our case study presented in Section 4, the size of process models was 40KB approximately, reaching 4GB with a hundred thousand processes. We expect, however, that there would be significant duplication in learned models; not all the processes would perform different operations, but several processes would have the same role in the system, thus generating similar function call traces. For example, typical distributed software for clusters would employ a tree-style network topology with varying tree heights for organizing each member process, where each node would have a different role depending on its depth in the tree. A leaf node process would only communicate with its parent process, while an internal node process with both its parent and child processes. The tree root process would be responsible for overall process management, responding incoming user requests, etc.

To derive a concise global model where these duplicated behaviors are removed, we categorize processes into groups where we can expect that their roles are the same inside each group. We infer the role of each process by the network connections established in its initializer unit. For example, in the tree-topology organization, the root process would establish connections by accepting requests from its children, while the leaf processes would do so by connecting to their parent. By looking at the call stacks that establish connections in the process initialization stage, we define the equality relation of processes, and derive a single model for equal processes. More specifically, we identify such call stacks in the initializer unit by locating calls to `bind`, `connect`, `listen`, and `accept`. Let $S(p)$ be the set of those stacks of process p. We call $S(p)$ the *process signature* and define the equality of process p and q by the equality of their signatures $S(p)$ and $S(q)$.

Figure 2 illustrates an example of process grouping in a master-worker system. The master process, which performs operations written in the nearby rectangle, forms a process group alone. The italicized lines, including listening to connections from clients and workers, denote the process signature calls for the master process. On the other hand, the three worker processes form a single process group, since they execute the same signature functions, i.e., connecting to the master.

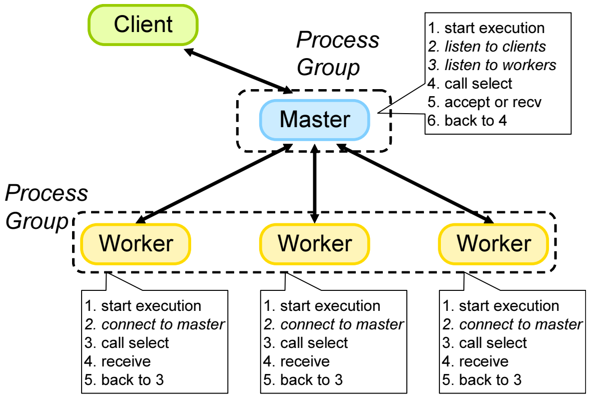


Figure 2. An example of process grouping in a master-worker system. Each of the master and worker processes performs the operations described in the nearby rectangle, where italicized lines denote the process signature calls.

Once the process models are categorized into groups, we merge the models in each group into a single process model as follows. First, for each initializer and finalizer model, we merge them by aggregating the trees and annotating each node by the mean probability of the original nodes. Next, for each handler model, we identify the matching model with the same associated connection as in the handler unit grouping in the process model derivation. If such a model is found, we merge them into a single handler model in the same way as the initializer and finalizer models. Otherwise, we copy the model to the resulting merged model as is.

3 Model-Based Fault Localization

Once the global model for a target system is derived, we deploy it to each local node so that we can perform the following model-based localization in a decentralized way. When a failure happens in the system, we compute a *suspect score* that quantifies the correlation of each execution unit in the traces with the observed failure by comparing the traces with the pre-deployed model. Our scoring algorithm described below gives high suspect scores to units whose behaviors are greatly deviated from the learned normal model. Finally, we gather the scores to a central location and report the scores sorted in a decreasing order to the problem analyst so that more suspicious parts of the program can be prioritized in a further localization process.

3.1 Suspect Score Calculation

We compute the scores by the following three-step decentralized process:

Step 1: Decomposing traces Decompose the traces into execution units in the same way as the model derivation.

Step 2: Finding the corresponding process model Find the process group with the same process signature. If no such corresponding process group is found, report the process as an anomaly to the analyst. If found, proceed to the next step.

Step 3: Finding the corresponding execution models For each trial unit, find the corresponding execution model in the process model as follows. For both the initializer and finalizer units, we locate the initializer and finalizer models, respectively. For each handler unit, we find the handler model with the same connection, where the equality of connections is defined in the same way as the model derivation phase. If no corresponding handler model is found, we mark the unit as an anomalous unit. If found, we compute its suspect score by comparing the unit with the found model.

For the anomalous processes and units whose corresponding models are not found in the above steps, we give the maximum suspect score of 1. For the other units, we compute their anomaly scores by the algorithm described below.

We consider the following functions more suspicious: those with higher probabilities of occurrence, but not called when a fault happens, and those with lower probabilities of occurrence, but called when a failure happens. Thus, our goal in designing the suspect score calculation is that it gives higher values to those more suspicious functions.

First, we construct a call tree from the given trial unit, u, in the same manner as the model derivation. Next, we compute *commonality* and *minimum difference* sets of the nodes in the model and trial tree. Let M and T be the sets of nodes that appear in the model and trial units, respectively. We define commonality, $M \cap T$, by the standard definition of the set commonality. We introduce minimum difference, $M \oplus T$, to filter out duplicate contributions to the suspect score calculation. For example, suppose that there is a call stack $f \rightarrow g \rightarrow h$ in a model, where g is called by f and h is called by g. In this case, if g is not called in a trial unit, h must not be called either. Since the absence of h is a direct effect of the absence of g, the former absence should not be of interest in assessing the difference between normal and anomalous executions. Based on this observation, we define the minimum difference as a set of nodes that appear only in either of the two sets, but excluding those nodes whose parent is also included in the minimum difference set. For example, in Figure 3, the commonality of the trees includes the nodes labeled with a, b, and c, and the minimum difference only includes the nodes d and e.

Next, we define an *effective node set*, E, as the union of the commonality and minimum difference sets, i.e., $E = (M \cap T) \cup (M \oplus T)$, and call the nodes in E the *effective nodes*. For every effective node $n \in E$, we compute the suspect score $\Delta(n)$ as follows:

$$\Delta(n) = \begin{cases} 1 - p(n) & \text{if } n \in M \cap T \\ p(n) & \text{if } n \in M \wedge n \notin T \\ 1 & \text{if } n \notin M \wedge n \in T \end{cases} \tag{1}$$

For example, if a function was called 90% of the times when the system was operating normally and was also called when a failure happens, we give Δ of 0.1 to that node. The node b in Figure 3 illustrates such a case. On the other hand, if that function was not called in the given trial unit, we give Δ of 0.9. This scoring scheme meets the design goal of suspect scores.

Finally, we define the suspect score $\Delta(u)$ for the given unit u as follows:

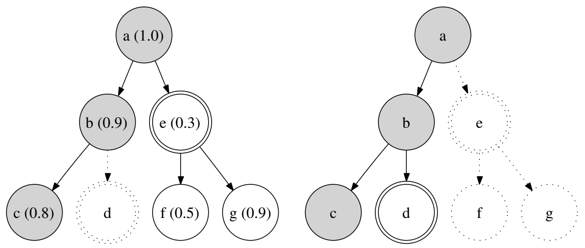
$$\Delta(u) = \frac{\sum_{n \in E} \Delta(n)}{|E|} \tag{2}$$

In other words, we use the average of the scores of all nodes in the commonality and minimum difference sets as the suspect score of the unit. Since we only consider the effective nodes, we avoid having non-interesting calls affect the overall scoring. For example, we compute the suspect score of the trial unit in Figure 3(b) as follows:

$$\begin{aligned} \Delta(u) &= \frac{\sum_{n \in E} \Delta(n)}{|E|} \\ &= \frac{\Delta(a) + \Delta(b) + \Delta(c) + \Delta(d) + \Delta(e)}{5} \\ &= \frac{0.0 + 0.1 + 0.2 + 1.0 + 0.3}{5} \\ &= 0.32 \end{aligned} \tag{3}$$

4 Preliminary Evaluation

To evaluate the effectiveness of the proposed approach, we have applied our prototype fault localizer to a known bug in a distributed job manager called MPD. MPD spawns a parallel job on a specified set of machines, monitors the job status, and returns the output



(a) Normal Model (b) Trial Unit

Figure 3. A sample normal model and its trial unit. In the left tree, the value in the parentheses of each node shows its estimated probability. Dotted edges and nodes indicate that such nodes or edges do not exist in that tree. Grey nodes indicate the commonality, while nodes with double lines the minimum difference.

of each process to the user. It is shipped with MPICH2, a standard implementation of the Message Passing Interface (MPI), and used by a wide variety of parallel programming users [10].

A user of the distributed computing platform called InTrigger reported a hang of MPD when he had tried to run a small MPI program. InTrigger is a large-scale computing platform consisting of six clusters distributed over Japanese universities and national laboratories. We applied our localization method to MPD running on the InTrigger platform, and have successfully identified an anomalous event that is highly correlated with the bug. The rest of this section describes the experimental setup and the fault localization result of the bug.

4.1 Prototype Implementation

To collect function call traces, we have implemented a tracer for C and Python programs as well as a non-blocking concurrent trace buffer pool. The buffer pool allows both traced processes and trace readers to access trace data in a concurrent, non-blocking fashion. Below, we describe their implementation details.

4.1.1 Trace Collection

Our current implementation supports tracing of function calls in C and Python programs as well as dynamic library calls. For tracing C programs, we currently use the compile-time function-call instrumentation available in the gcc compiler, which requires recompilation of the traced program by the gcc compiler. We are planning to use binary instrumentation tools, such as Dyninst [6], for greater flexibility.

For tracing Python programs, we use the debugging API, `sys.settrace`, that is available in the official Python implementation. We require almost no modifications to the traced program; we use our own Python code that sets our tracing functions to be invoked each time when the traced process makes calls and returns, and then calls the original starting function of the target program.

For tracing dynamic library calls, we use the library preloading mechanism available on standard Unix and Linux systems so that our library wraps the target library calls. This technique requires no modifications in the traced program, but setting an environment variable, `LD_PRELOAD`, to include our tracer library. The wrapper functions, upon called by the target, generate trace entries before and after calling the real library functions.

4.1.2 Non-Blocking Concurrent Trace-Buffer Pool

To make a trace buffer accessible from both a traced process and trace readers, we inject a shared library into the process by the dynamic library preloading mechanism. The library, upon loaded, allocates a shared memory region of specified size, and divides it into sub-buffers. Each sub-buffer is in either *free* or *written* state, enqueued to either free queue or written queue, respectively. The traced process records its traces into free buffers, while trace readers consume the traces from written buffers. Figure 4 illustrates an example case where a buffer pool is used by a single traced process and three reader processes. By exploiting the library preloading mechanism, we again require no modification of the traced program itself.

The trace library initially adds the sub-buffers to the free queue for write accesses from the target process. A target process, instrumented to generate function call traces by the abovementioned methods, obtains a free buffer from the free queue in the pool, and starts executing its functions, appending call traces to the buffer. Each time when the traced process finds the current buffer becomes out of space, it gets the next available free buffer from the pool. Simultaneously, multiple trace reader processes can attach to the pool and get the written buffers for read access. Upon finishing reading the buffer, it returns the buffer to the pool, which then becomes available as free buffers.

When a traced process attempts to obtain a free buffer, but no more such buffers are currently available, it gives up generating traces for a specified number of calls and returns, instead of blocking on a free buffer becoming available. It retries to get a free buffer after the specified count of calls and returns. This non-blocking scheme aims to minimize the perturbation to the traced process, while sacrificing the completeness of the trace.

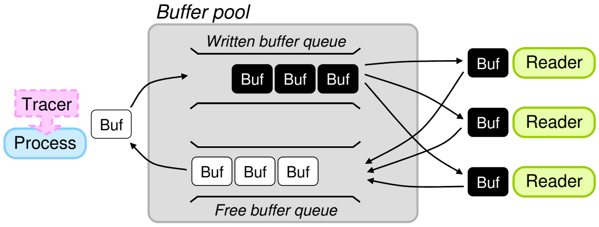


Figure 4. An example usage scenario of the trace buffer pool.

4.2 Partial Message Receive Bug in MPD

MPD manages each node by running a daemon process that is connected each other by a ring-topology network. When a new parallel job is submitted to the system, it spawns a specified number of processes by forwarding the job information over the ring network. The network is also used to coordinate the ready state of each process to start execution.

The reported hang occurred when the user submitted a small MPI program using the MPD version 1.0.5p4 running on multiple distributed clusters. Although the MPI program is a very small test program that runs flawlessly within a single cluster, using multiple clusters caused the program not to start, but apparently to be hanging up during its job startup stage. Further examination by the user through printf-style message logging revealed that one of the MPD job manager daemons erroneously closed a connection in the ring-topology daemon network, causing all the daemons to infinitely wait for a particular message that should have been sent over the ring connection. The reason of the connection reset turned out that one of the calls to `recv` function in the socket API silently ignored its return value. The call expected to receive eight bytes every time when a message arrives to the connection, but in fact it sometimes received partial messages, which in turn caused the daemon to close the connection, because it erroneously decided that an error occurred in the system. Of course, the assumption does not necessarily hold on distributed environments, yet, since MPI is mainly used in tightly-connected single-site clusters, the bug had not been reported before.

This particular fault exhibits several common properties that make fault localization particularly difficult in such a large-scale environment. First, it is non-deterministic: in some runs, the daemons always received completely-formed messages, allowing jobs successfully to be started even on multiple clusters. In fact, the greater the number of nodes, the more often the fault happened, requiring scalable localization techniques. Second, since it is a timing-related bug, use of an interactive debugger, if possible, would significantly reduce the chance to reproduce the bug, although debugging of as little overhead as printf-style message logging still allowed the bug to occur. Third, it is not a fail-stop, but a silent bug. Although the ring topology was not operating correctly after the bug occurred, each daemon was still in a normal state, waiting on the event-processing loop. While examining a call stack of a failed process is an effective debugging technique in many cases, it would not help reveal the root cause in this case; the stack trace would look normal since waiting on the event loop is a legitimate operation.

4.3 Model Derivation

We generated the global model from normal traces of MPD runs on a single cluster as follows. First, we traced MPD on seven different configurations using a simple MPI program called CPI and the NAS Parallel Benchmark [19] as sample parallel jobs. CPI, shipped with the MPICH itself, calculates the value of π by a Monte Carlo method. For the buffer pool on each node, we allocated ten sub-buffers of 1MB, or 10MB in total. We obtained per-process function call traces using an online trace reader that saves the contents of the written trace buffers to its local disk. Next, on each local node, we generated a process model for each trace using our prototype model generator. Next, we gathered the process models into a central location and generated the global model. Below, we describe the detailed results of model derivation as well as performance overhead caused by function tracing.

4.3.1 Process Model Derivation

Table 1 lists the seven configurations and their results on process model generation. We executed traced MPD on different numbers of nodes, each of which hosted a single MPD process. The three columns of time, trace size, and model size list the averaged values of all the nodes. Because we implemented the model derivation by a Python program, and thus derived process models are Python objects, we measured their sizes by serializing them to binary data. We see that, while the execution times vary significantly, the sizes of resulting process model are very similar, suggesting that MPD had exhibited repetitive function call behaviors and our modeling efficiently encoded the behavior without substantial duplicated information.

To study the performance impact on the application programs by MPD tracing, we have compared their execution times with and without our tracing enabled. In Figure 5, the y-axis shows the relative performance

Table 1. Process model generation configurations and results.

Application	# processes	Time (sec)	Trace Size (KB)	Process Model Size (KB)
CPI	58	0.95	949.2	43.8
CG Class C	4	127.75	448.8	43.4
IS Class A	8	3.29	483.7	43.7
CG Class C	16	58.31	566.9	43.5
MG Class B	32	2.16	682.9	43.7
LU Class D	32	1782.85	939.7	43.6
BT Class D	49	1223.16	1395.3	43.6

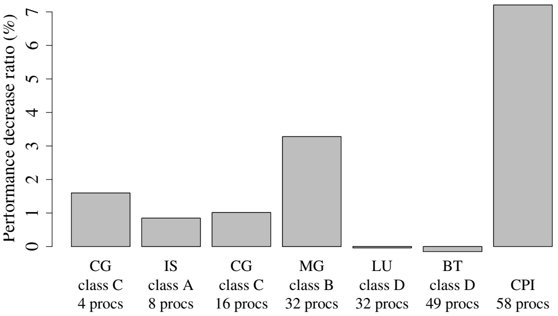


Figure 5. Performance overhead due to function call tracing.

when executed under our tracer compared to normal executions. The overhead ranges from approximately 0% to 7%; while such a small overhead does matter for extremely performance critical systems, we expect that such small perturbation would be affordable in many cases, considering the effectiveness of our fault localization presented below.

4.3.2 Global Model Derivation

We gathered all the 199 process models into a central location, and generated the global model including the overall behaviors of MPD. The resulting model was 171KB in size with only three different process groups. Although we aggregated such a large number of process models with different runs, this result again suggests the space efficiency of our model derivation.

4.4 Localization through Suspect Score Computation

We obtained a set of per-process trial traces by reproducing the bug on three geographically-distributed clusters in InTrigger, called hongo, chiba, and okubo.

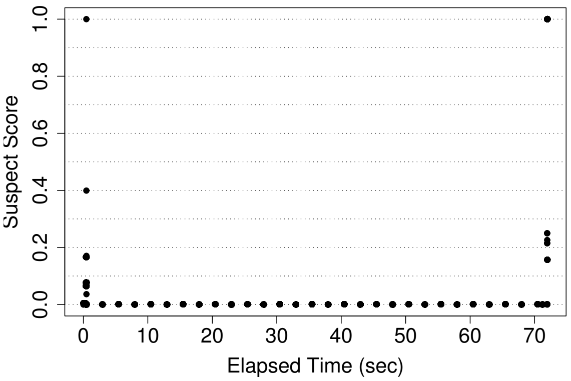


Figure 6. Suspect scores of the execution units in the trial traces.

We started traced MPD with the sample MPI program CPI on 78 nodes spanning the three distributed clusters, and decided that the system hung up after observing no output for approximately 70 seconds. We stopped the system by killing all the daemons. Out automated analysis decomposed the trial traces into execution units and computed the anomaly scores.

Figure 6 shows the scores of all the units plotted at their timestamps. We see that the beginning and ending of the run had significantly deviant behaviors, while the rest did not exhibit any interesting behaviors. We see that the most interesting unit is the one with the maximum score near the beginning of the run, since we expect that the earlier a deviant behavior occurs, the more likely it would be correlated with the failure. We find that the particular unit is a handler unit that was notified of a connection-reset event. The reason of the high suspect score of the unit is that the reset event had never been observed in the normal traces. From the trace entries before the unit, we found that the other peer of the connection was running on node hongo102 in the hongo cluster, while the unit itself was on node chiba121 in the chiba cluster. Another high-score duration before the end of the run turned out to be caused by our abrupt killing of the daemons.

To identify why the connection was reset between chiba121 and hongo102, we manually examined the MPD source code and the trace entries in hongo102 around the time when the reset happened. We found that hongo102 closed the connection in function `handle_lhs_input` of `MPDMan` class, and that the function closed it because a call to `recv_dict_msg` of `MPDSock` failed. The trace entries for the calls from `recv_dict_msg` included a call to `recv` that actually received only three bytes of data from the connection, while `recv_dict_msg` expected eight bytes. As a result, because of the partial-message receive bug, the execution of `recv_dict_msg` failed, causing `handle_lhs_input` to close the connection.

This case study suggests the effectiveness of our localization support. Although we were not able to pinpoint the buggy `recv` call, our suspect score ranking identified as the most suspicious event the connection close event observed by the connection peer process. Without our automated trace analysis, we would have needed to examine all the traces of 78 nodes accounting for the complete 70-second run. Our localization analysis narrowed the localization only to the traces of two nodes accounting for less than a second, significantly reducing the manual fault localization burden.

5 Related Work

Most of previous approaches to fault detection and localization in distributed systems use tracing with some variation. For example, trace-based automated detection of performance and logic bugs proposed in, among others, [2, 4, 7, 12, 14–16, 20], finds anomalies in large volume of traces with various statistical and machine-learning techniques. In the rest of this section, we elaborate on the similarities and differences with previous projects on fault localization and other closely-related areas.

Identifying distributed control and data flows is a key algorithm in many of the trace-based automated performance and logic debugging approaches. Barham et al. presented a flow analysis method that uses a user-written application-specific event schema and instrumented OS kernel and other middleware layers [4]. The schema is a set of rules to separate and join individual events observed in distributed components. Chen et al. and Kiciman et al. presented root-cause identification approaches that are also based on flows in distributed environments [7, 8, 12]. Their approaches assume that the system under observation has naturally-observable user request flows, such as RPC-based systems. For example, an algorithm targeted to HTTP-based distributed systems uses HTTP request logs to recover user request flows in web-server farms. Another algorithm assumes that the target system uses a high-level component framework, such as J2EE, and modifies an underlying framework implementation to record component interactions. To find root causes of a failure, they correlate the failure with particular components by analyzing recovered flows with several statistical techniques, including cluster analysis, decision trees, and probabilistic context-free grammar. Mirgorodskiy presented another trace-based flow-recovering algorithm that requires little human burden by automated binary instrumentation across node boundaries [14]. Similar to us, they use function call traces, and quantify the differences of distributed function call flows to find anomalous calls. They also use a filtering rule that removes duplicated contributions from child nodes. Reynolds et al. proposed an approach to assisting system developers in detecting unexpected system behaviors [16]. Similar to our model generation, their approach first infers from test runs the expected program behaviors that include interaction of distributed components, and generates their textual representation. The auto-generated expectation, which the user can extend for more accurate analysis, is checked against the traces of trial runs. Aguilera et al. [2] presented a statistical algorithm that requires no a-priori knowledge on the target system, while trading off the accuracy of the recovered flows.

The key difference between these previous approaches and ours is that while their flow-based algorithms assume centralized processing, we designed our localization algorithm to be mostly decentralized. As shown by Roth et al. [18], decentralized processing is essential to work at the scale of today's HPC systems, such as the 106,496-node Blue Gene/L at Lawrence Livermore National Laboratory and the 655-node Tsubame supercomputer at Tokyo Institute of Technology. Another notable difference is that flow-based approaches require to determine correlation of message send and receive operations by either message counting as in [14] or tagging as in [7, 8, 12]. While counting is relatively simple to implement, it cannot be used for datagram-based connections. Message tagging works for both stream- and datagram-based connections, but the perturbation due to embedding a tag into each message can be too large to catch non-deterministic, timing-related bugs.

Research projects on fault localization that do not rely on distributed flows include Mirgorodskiy et al. [15], Zheng et al. [20], and Arnold et al. [3]. Similar to us, Mirgorodskiy et al. proposed a fault localization method using function call traces [15]. To localize faults, it exploits an observation that in a SPMD-style distributed program, its member processes should be-have similarly to each other. It detects anomalous processes by finding outliers in function execution time profiles of the target processes. However, such assumption does not necessarily hold in distributed systems. For example, the master process of a master-worker system would always be considered an anomaly since it would have no other similar process. To compensate such natural differences, their method can also use previous known-normal traces; if previous traces include one that is similar to outlier traces, they consider the outliers normal. Our modeling is similar to this method since we also use previous data to localize faults. However, ours is more space efficient since we do not simply keep previous data but a concise representation of normal behaviors as exeuction models. Furthermore, it is more scalable. Once models are derived, the analysis for fault localization is completely decentralized since it requires no network communication, while they need to exchange the time profiles among all the processes to compare the traces of each process.

Zheng et al. also presented an anomaly detection algorithm based on the node-similarity assumption [20]. Unlike Mirgorodskiy et al., they use standard performance metrics, such as CPU and memory load. We could also improve the effectiveness of our approach by using such metrics as well. For instance, such bugs as deadlocks might manifest themselves as significantly lower CPU load. However, their method can only reveal process-level anomalies, unlike our function-level analysis that can identify anomalous functions.

Arnold et al. proposed a bug detection method using stack trace sampling [3]. Similar to us, their primary focus is scalability with an increasing number of nodes. Thanks to MRNet, a tree-based overlay network [17], they achieve low-latency collection of call stacks from thousands of processes; we could also make use of such a scalable overlay network for gathering process models. Unlike us, they use a sampling-based analysis, which makes detecting rare anomalous behaviors difficult.

Another related research area is the scalability of performance analysis in large-scale HPC systems [9, 18]. Roth et al. presented a scalable performance analysis framework based on MRNet [18]. They gather performance profiles at run time by construct a tree network of distributed nodes using MRNet. Their highly decentralized analysis framework allowed the performance bottleneck search by Paradyn [13] to be possible with moderate CPU and network load even in more than a thousand processes, while a centralized analyzer could not handle such a large number of processes. Tree-based overlay, like MRNet, would also be useful in our global model derivation, though we have not yet seen bottlenecks in gathering process models.

Geimer et al. proposed a parallel algorithm for finding communication patterns with sub-optimal performance [5, 9]. They use the same set of nodes as the target system, and attempt to discover inefficient communication patters in a scalable way by replaying each communication event on the same node and identifying distributed message correlation. We could have used such a parallel correlation-based technique in our model derivation phase as well. For example, while the current modeling assigns a single model to each connection, we could have differentiated function calling patterns by using the call stack of the sender as a key. By doing so, we could have generated a model for a pair of a unique connection and its sender, which would improve the accuracy of the resulting model. However, it in turn requires to find distributed message correlation by such methods as message counting and tagging. Since the overhead incurred by message correlation can be too large to detect timing-related bugs, the effectiveness of correlation-based analysis applied to our problem domain remains unclear and a subject of future work.

6 Conclusion

We presented our model-based fault localization technique that aims to help the human analyst narrow down the manual localization process into a small fraction of the whole system. Our method consists of two parts: pre-fault model generation and model-based anomaly detection. The first part collects function call traces from each process and derives an execution model that reflects the function calling behaviors of the whole system. When a failure happens, we compute the anomaly score of each execution unit in the trial traces by comparing it against the derived model. The anomaly score, ranging from 0 to 1, quantifies how likely the execution unit is correlated with the fault. Our claim is that the analyst can substantially reduce the manual localization burden by prioritizing the execution units with higher anomaly scores. Our preliminary experiment with a distributed job manager supported this claim: our method narrowed down a bug finding process of a 70-second faulty run on a 78-node distributed platform into just sub-second behaviors on two nodes.

We have several remaining issues to explore. First, we will explore online approaches to model derivation and anomaly score calculation. Our current method dumps function call traces into local disks by running the online trace dumper in the background of the target process. While this scheme was effective for the experiment with MPD, the trace size dumped to the local disk could be too large to keep for longer-running programs. We will explore online modeling and anomaly score calculation to keep the space overhead minimum. Second, we will apply our technique to localization of faults in different distributed systems. Specifically, for each different target system, we are planning to investigate 1) whether the assumed event-driven architecture holds or not, 2) how concise our modeling can encode its behaviors, and 3) how effective the model can be effective in localizing faults on the system.

Acknowledgments This research is supported in part by the MEXT Grant-in-Aid for Scientific Research on Priority Areas 18049028 and the JSPS Global COE program "Computationism as a Foundation for the Sciences."

References

[1] Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 2: System Programming, Sept. 2007.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 74–89, 2003.

[3] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Debugging Large Scale Applications. In *International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–10, 2007.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 259–272, 2004.

[5] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. N. Wylie, and B. Mohr. Automatic Trace-Based Performance Analysis of Metacomputing Applications. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–10, Long Beach, CA, March 2007.

[6] B. R. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[7] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[8] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 595–604, June 2002.

[9] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable Parallel Trace-Based Performance Analysis. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI User' s Group Meeting*, volume 4192, pages 303–312, 2006.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sep 1996.

[11] Intel Corporation. Intel 64 and IA-32 Arhictectures Software Devleoper's Manual Volume 1: Basic Architecture, Aug. 2007.

[12] E. Kiciman and A. Fox. Detecting application-level failures in component-based Internet services. *IEEE Transactions on Neural Networks*, 16:1027–1041, Sept 2005.

[13] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[14] A. V. Mirgorodskiy. *Automated Problem Diagnosis in Distributed Systems*. PhD thesis, University of Wisconsin-Madison, 2006.

[15] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem Diagnosis in Large-Scale Computing Environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06)*, Tampa, Florida, November 2006.

[16] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 115–128, San Jose, CA, May 2006.

[17] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of ACM/IEEE Conference Supercomputing (SC'03)*, page 21, 2003.

[18] P. C. Roth and B. P. Miller. On-line Automated Performance Diagnosis on Thousands of Processes. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 69–80, New York, NY, USA, March 2006.

[19] P. Wong and R. F. V. der Wijngaart. Nas parallel benchmarks version 2.4. Technical Report NAS-02-007, NASA Ames Research Center, 2002.

[20] Z. Zheng, Y. Li, and S. Zhiling Lanteri. Anomaly Localization in Large-Scale Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'07)*, pages 322–330, Austin, Texas, Sep 2007.