PInfer: Learning to Infer Concurrent Request Paths from System Kernel Events

Hongteng Xu
School of ECE
Georgia Institute of Technology
Email: hxu42@gatech.edu

Xia Ning
Indiana University-
Purdue University Indianapolis
Email: xning@cs.iupui.edu

Hui Zhang, Junghwan Rhee, Guofei Jiang
Department of Autonomic Management
NEC Laboratories America
Email: {huizhang,rhee,gfj}@nec-labs.com

***Abstract*—Operating system kernel-level tracers are popularly used in the post-development stage by black-box approaches. By inferring service request processing paths from kernel events, these approaches enabled system diagnosis and performance management that are application-logic aware. However, asynchronous communications and multi-threading behaviors make request path patterns dynamic on the kernel event level; this causes previous methods to focus on either software instrumentation techniques or better statistical inference models.**

In this paper, we propose a novel learning based approach called PInfer that infers request processing path patterns automatically with high precision. PInfer first learns dynamic event patterns of inter-thread and intra-thread service processing from the training data of sequential requests. On the testing data containing concurrent requests, PInfer infers individual request processing paths by effectively solving a graph matching problem and a generalized assignment problem based on the learned patterns. We have implemented our approach in a proprietary system performance diagnosis tool, and present performance results on 40 sets of kernel event traces. PInfer achieves on average 65% precision and 85% recall for profiling concurrent request processing paths.

***Keywords*-request processing path; dynamic event patterns; learning based approach;**

I. Introduction

In large-scale production cloud computing infrastructures, system diagnosis (e.g., investigation of resource and performance issues) is a realistic and important problem due to frequent incidents at their scales. Diagnosis with low-level system events (e.g., system calls) has been studied in many approaches [1], [2], [3], [4], [5], [6], [7], [8], [9]. These are black-box approaches which do not rely on application software's internal knowledge, and are desirable for service operators. They can work without source code at hand, look into all components with a system-wide view, and pinpoint the component(s) responsible for the issues.

To enable diagnosis of end-to-end system behaviors, a necessary feature is to model its high-level behavior based on low level system events. Networked service systems process a service request from users and we use the term, *a request-processing path* [6], to represent the series of system activitie across all necessary components. More precisely it is defined as all system events, including network and thread activities, starting from the user request received at the front-end, until the final response sent back to the user. Figure 1 shows a request-processing path going through multiple components; it starts from a web server thread (left most) which receives an external client request; it then goes through an acceptor thread, a selector thread, and a scheduler thread in the application server, before the request is dispatched to a worker thread; lastly, the request processing path finishes when the web server thread replies to the client after exchanging a few messages with the application server worker thread.

It is challenging to profile concurrent request paths in modern service systems with complex asynchronous and multi-threading behaviors [6]. Specifically, those behaviors can be summarized as follows:

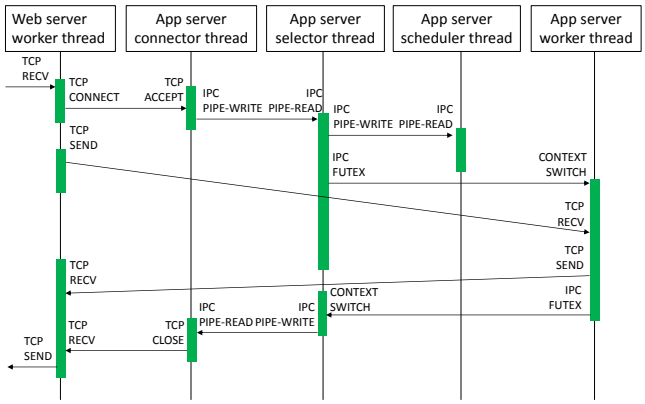

Figure 1. A request processing path example going through five distributed application threads in a multi-tier service system.

Table I
MULTIPLE REQUEST PROCESSING PATTERN CONFIGURATIONS SUPPORTED IN *Jetty* [10] APPLICATION SERVER.

1	Synchronous request.
2	Asynchronous request.
3	Asynchronous request with new thread created.
4	Asynchronous request without return.
5	Asynchronous request with new thread and without return.

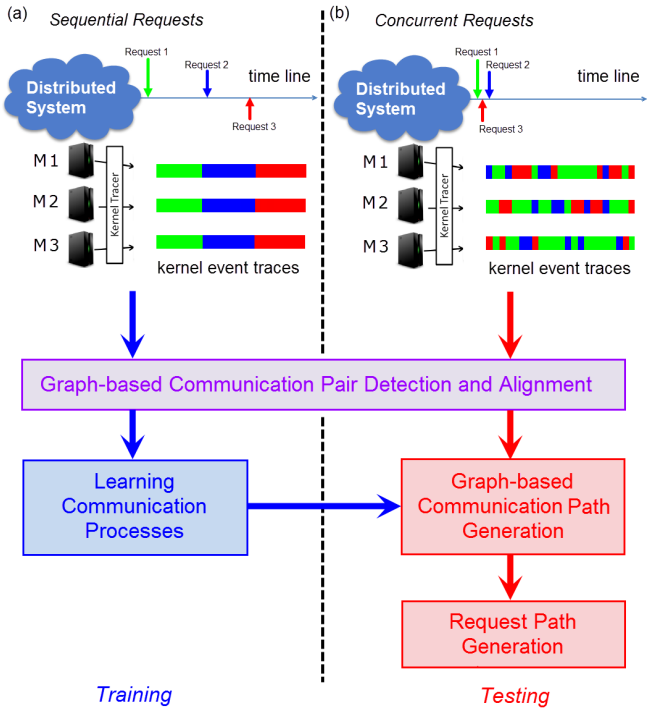

Figure 2. PInfer scheme. (1) In the training phase the communication processes are learned. (2) In the testing (inference) phase, we generate communication paths and then generate request processing paths hierarchically. The training data is a set of sequential request paths without overlaps while the testing data contains a lot of concurrent requests, whose paths are interleaved with each other.

- *Cooperative behavior*: a request is processed by more than one thread cooperatively. For instance, in Figure 1, three threads work together to accept and schedule a request before another worker thread performs sending and receiving packets;
- *Collective behavior*: in a threading pool model, there are a group of threads/processes taking a similar role and handling requests alternatively in a pool. In Figure 1, the pool of worker threads in an event-driven web server (such as *nginx* [11]) is such an example.

These multi-threading behaviors make request paths dynamic in asynchronous multi-threaded server systems as a request path may involve multiple threads in a non-deterministic way; due to the lack of application level request information in kernel events, the transitions between those threads during the request handling are usually hard to be traced precisely without using application-level instrumentation tracing. Besides, many service systems have multiple configuration options, each of which will define a different request processing pattern on kernel event level. Table I shows an example of the five configurations supported in *Jetty* [10], a popular open-source application server.

In this paper, we take a learning based approach, called PInfer, to automatically learn and infer individual request processing paths from massive amount of OS kernel events containing concurrent requests. The principle of PInfer is that although request paths are various, the communication processes between threads in these paths have stable pairwise patterns. Inspired by this fact, PInfer divides original inference problem into two phases: inferring communication processes in request paths, and then generating request paths according to the communication processes. This strategy simplifies the inference problem greatly. Note the communication event definitions are in terms of pairs, and are not limited to network events; the details are in Section III.

As shown in Figure 2, PInfer has two stages. (1) In the training stage, event traces of sequential requests are collected, and PInfer learns the pairwise patterns of communication processes from the traces. (2) In the testing (inference) stage with event traces of concurrent requests, PInfer infers end-to-end communication paths by solving a generalized assignment problem based on the learned patterns. The request processing paths can be generated based on the inferred communication paths. The inferring communication path process is composed of three steps: communication pair detection, alignment, and stitching. In these three steps PInfer solves two graph-based problems: the detection and the alignment of communication pair is formulated into a graph matching problem, while the stitching of communication pair can be achieved by solving a generalized assignment problem on a bipartite graph.

The contributions of this paper are summarized as follows:

- A new learning based approach is proposed for solving the end-to-end request profiling problem. Previous methods focused on either application instrumentation techniques or developing better statistical inference models. We instead take a very different machine learning methodology to radically simplify the problem.
- We implemented PInfer in a proprietary system performance diagnosis tool named called CLUE [12], applicable to IT systems in multi-core servers on major OS platforms including Linux (Redhat, Fedora), Unix (HP-UX), and Windows (Windows Server 2008).
- We conducted an extensive evaluation of PInfer on 40 sets of kernel event traces collected from a multi-tier service system with different service workloads, system setups, and request pattern configurations. PInfer can achieve on average 65% precision and 85% recall on profiling concurrent request processing paths.

The rest of the paper is organized as follows. In Section II we present the related work. Section III describes background of our work and notations of terms. Section IV and V present PInfer's communication process learning algorithm and request processing path inference algorithm. Section VI gives the analysis of PInfer's proposed algorithms. We present the evaluation results in Section VII, and conclude this paper in Section VIII.

II. Related Work

Several black-box approaches target capturing the path and the timing of an individual service request across the components of a multi-tiered system. Approaches such as Project5 [13] and WAP5 [14] accept imprecision of probabilistic correlations. Project5 proposes a nesting algorithm which assumes RPC-style (call-returns) communications. WAP5 infers causal paths for wide-area systems in a per-process granularity using library interposition.

Precise black-box approaches such as BorderPatrol [15], vPath [6], and PreciseTracer [16] build request traces for specific protocols or application threading models. BorderPatrol isolates and schedules events or requests at the protocol level to precisely track service requests with the explicit knowledge of diverse protocols used by multi-tier services. For commercial components or heterogeneous middleware, BorderPatrol needs to provide customized protocol processors and requires specialized knowledge than pure black-box approaches. The vPath system continuously logs in a virtual machine monitor regarding which thread performs communication system call over which TCP connection, and makes assumptions about the threading model of distributed system such as synchronous communication among components of the system and a single thread handling all the messages common to one request. PreciseTracer [16] offers an online request tracing system. It first reconstructs network communication events into causal paths, then classifies these causal paths into different patterns according to their shapes. Its final presentation is an macro-level abstraction, dominated causal path patterns, to represent repeatedly executed causal paths that account for significant fractions. All these black-box approaches take only a small subset of system/network events into their analysis, and rely on explicit request-reply communication patterns.

In networking area there are studies on network event sequence analysis and applications. For example, Meng et al [17] proposed a novel approach that automatically captures the behaviors hidden in massive event sequences using a mixture Markov Chain model. While those approaches identify significant patterns such as flows and host communication patterns, their outputs are statistical models and do not address the full path inference problem addressed in this paper.

III. Background and Notations

A. Operating System Kernel Events

As a core component of a computer system, an OS kernel provides the lowest-level resource abstraction layer for application software. Typical examples of kernel events include system calls from processes, scheduling events, interrupts, I/O operations, and locking operations.

In this paper, we denote a *kernel event* as $e = (t,\ \boldsymbol{v})$, where t is the time stamp of the event e, the vector $\boldsymbol{v}$ represents the parameters of the event, including source and destination IP, thread ID, CPU ID, and port number, etc.

Table II
List of communication pairs.

	Intra-Communication Event Pair
1	⟨FUTEX, PRESUME⟩
2	⟨CLONE, PRESUME⟩
3	⟨UNIX_STREAM_SEND, UNIX_STREAM_RECV⟩
4	⟨PIPEWIRTE, PIPEREAD⟩
	Inter-Communication Event Pair
1	⟨TCP_CONNECT, TCP_ACCEPT⟩
2	⟨TCP_SEND, TCP_RECV⟩
3	⟨TCP_CLOSE, TCP_RECV⟩
4	⟨TCP_SHUTDOWN, TCP_RECV⟩
	Starting Event of a Request
1	⟨−, TCP_RECV⟩ with an external destination IP
	Ending Event of a Request
1	⟨TCP_SEND, −⟩ with an external destination IP

B. Communication Event Pairs

We can categorize various kernel events into two classes: the communication events and the rest. An important property of communication events is that they always appear in pairs. We denote a *communication pair* as $C = \langle e^b,\ e^s \rangle$, where e^b and e^s are the starting communication event and the ending communication event of the pair, respectively. we further categorize communication pairs into two classes: *intra-communication pairs*, the pairs in which e^b and e^s are generated in the same machine (e.g., shared memory based inter-process communication events), and *inter-communication pairs* such as TCP network events.

Table II lists a set of example communication pairs. For example, a FUTEX unlocking system call by a thread and the caused PRESUME context switch event on another thread waiting for the unlock forms an intra-communication pair defined in our scheme; a pair of TCP_CONNECT and TCP_ACCEPT network events for a TCP connection setup is another inter-communication pair defined in our scheme.

C. Request Processing Paths

Denote a sequence of kernel events triggered by K requests as $\mathcal{S} = \{e_1, e_2, ..., e_N\}$. We aim to infer a *request processing path* for each request and decouple the sequences of kernel events from $\mathcal{S}$ that correspond to different requests; a kernel event sequence for a request is a request processing path. For the kth request, its request processing path is denoted as $\mathcal{P}^k = \{e_1^k, e_2^k, ..., e_{N_k}^k\}$, which contains the whole process of the request. Furthermore, the communication pairs within a request processing path construct a *communication path*, denoted as $\mathcal{C}^k = \{C_i^k\}_{i=1}^{M_k}$. Here C_i^k is the ith communication pair in the kth request processing path. Figure 1 illustrates a typical request processing path, where a communication pair consists of two events connected by

arrows crossing threads and the rest events are represented as green bars in threads.

IV. PInfer Learning Scheme

In the learning phase, PInfer models communication path patterns in the processing paths of sequential requests. That includes discovering the starting and ending events of request processing paths, detecting communication event pairs, and learning the pairwise relationships among communication event pairs.

A. Communication Event Pair Detection and Time Stamp Alignment

For a request processing path, we assume the kernel event types of starting and ending communication events are defined by domain knowledge. Typically, those events are the network events for the requests and replies to customers outside of the traced system.

For the rest of communication events in the path, given a starting event $e^b = (t^b,\ \boldsymbol{v}^b)$, we can find a candidate ending event $e^s = (t^s,\ \boldsymbol{v}^s)$ in the time window $(t^b - \Delta t_1, t^b + \Delta t_2]$ by matching their parameters v^b and v^s. We set $\Delta t_1 = 0$ to detect intra-communication event pairs so that it guarantees the causality between intra-communication events (i.e., the time stamp of an ending event should be later than that of its paring starting event). For inter-communication event pairs, its causality is influenced by time alignment, so we set $\Delta t_1 > 0$ to allow asynchronization among different machines.

The detected inter-communication event pairs can help align event time stamps across different machines. This is especially helpful in today's data centers with network latency at sub-milliseconds level [18]. In a system of D machines $\{M_d\}_{d=1}^{D}$, for each pair of machines M_d and $M_{d'}$ ($d \neq d'$), ideally the clock of M_d should be synchronized with that of $M_{d'}$ via the following linear model:

$$t_{d'} = a_{dd'} t_d + b_{dd'} \tag{1}$$

where t_d and $t_{d'}$ are the time clock of M_d and $M_{d'}$, respectively, $a_{dd'}$ and $b_{dd'}$ are the frequency shift and the time shift of M_d with respect to $M_{d'}$. Suppose that there are $N_{dd'}$ inter-communication event pairs between M_d and $M_{d'}$ and the starting events are on M_d and the ending events are on $M_{d'}$. We can learn the clock frequency shift and the time shift by minimizing the time interval between the starting and the ending events.

In the event pair detection step, it is possible to find multiple ending events corresponding to one starting event. A greedy strategy to solve this issue is to select the ending event with the closest time stamp. However, the notion of "closest" relies on clock synchronization. On the other hand, the time stamp alignment depends on the correct match of communication event pairs. In addition, detecting communication event pairs one by one will increase the risk of error propagation since the detection of a wrong communication event pair will impact the detection of other pairs in the consecutive steps.

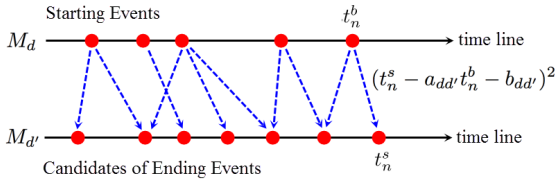

Figure 3. The graph matching model for communication event pair detection and alignment. The starting events of inter-communication event pairs and the candidate ending events (red points) construct two graphs, respectively. The blue dot lines correspond to potential connections between events, whose weights are errors of time alignment. The connections (inter-communication event pairs) are achieved via graph matching, which minimize the errors of time alignment.

B. A Joint Graph Matching Solution

Instead of an iterative method, we propose to solve the communication event pair detection and time stamp alignment problems simultaneously. In specific, we propose a bipartite graph matching scheme. As Figure 3 shows, for each starting event on machine M_d, we can find multiple candidates ending events from the event sequence on $M_{d'}$. We connect the starting events e_n^b and their corresponding ending event candidates e_n^s with a directed edge with weight $(a_{dd'}t_n^b + b_{dd'} - t_n^s)^2$. ($a_{dd'}$ and $b_{dd'}$ can be learned simultaneously, which will be discussed later).

Denote the set of starting events on machine M_d as $\{e_m\}_{m=1}^{M}$ with time stamps $\{t_m\}_{m=1}^{M}$, and the set of ending events candidates on $M_{d'}$ as $\{e_n\}_{n=1}^{N}$ with time stamps $\{t_n\}_{n=1}^{N}$. The connection between communication events e_m and e_n is represented by a binary indicator as follows.

$$x_{mn} = \begin{cases} 1, & \langle e_m, e_n \rangle \text{ is a pair,} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

We propose to conduct communication event pair detection and time alignment simultaneously by solving the following optimization problem.

$$\begin{aligned} \min_{x_{mn}, a_{dd'}, b_{dd'}} & \sum_{m=1}^{M}\sum_{n=1}^{N} x_{mn}(t_n - a_{dd'}t_m - b_{dd'})^2, \\ s.t.\ \ & x_{mn} \in \{0,1\}, \\ & \sum_{n=1}^{N} x_{mn} = 1,\ \ m = 1,..,M, \\ & \sum_{m=1}^{M} x_{mn} \leq 1,\ \ n = 1,...,N, \\ & x_{mn}(a_{dd'}t_m + b_{dd'} - t_n) \leq 0. \end{aligned} \tag{3}$$

The objective function in Equation (3) represents the weighted sum of matching errors between all starting events and their true corresponding ending events. The first constraint on x_{mn} corresponds to the definition in Equation (2). The second constraint ensures that each starting event is connected to exactly one ending event, given that the connection indicator is binary. The third constraint ensures that each ending event candidate is connected to at most one starting event. The last constraint guarantees the causality relation among communication events such that the ending events appear after their corresponding starting events.

The problem in Equation (3) is a special example of graph matching problem, which can be solved by iterative closest point (ICP) method [19], [20], and alternating minimization approaches. Algorithm 1 describes the details of the solution.

Algorithm 1 Graph-based Communication Event Pair Detection and Time Alignment

Input: Event sequence $\mathcal{E}$ from a system having D machines $\{M_d\}_{d=1}^{D}$, Δt_1, Δt_2.
Output: Aligned event sequences and communication event pairs.
for $d = 2 : D$ **do**
  Find communication starting events $\{e_n\}_{n=1}^{N}$ in M_d based on domain knowledge.
  for $d' = 1 : d-1$ **do**
    Find ending event candidates s $\{e_m\}_{m=1}^{M}$ in $M_{d'}$ based on domain knowledge.
    $a_{dd'} = 1$, $b_{dd'} = 0$.
    repeat
      Fix $a_{dd'}$, $b_{dd'}$ and solve (3) for x_{mn}.
      Fix x_{mn} and solve (3) for $a_{dd'}$, $b_{dd'}$.
    until convergence
    For all the events on $M_{d'}$, align their time stamps by $t := a_{dd'}t + b_{dd'}$
    Detect inter- and intra-communication event pairs with time stamps aligned.
  end for
end for

The domain knowledge described in Algorithm 1 is on the parameter matching conditions between communication event pairs; these conditions reflect the temporal order of the patterns of pairs, such as an invoking event (e.g., connect) preceding an invoked event (e.g., accept). A list of the conditions is as follows:

- For SETRQ (or CREATE), find PRESUME having larger time stamp and same source IP and thread ID.
- For UNIX_STREAM_RECV (or PIPEWRITE), find UNIX_STREAM_SEND (or PIPEWREAD) having larger time stamp and same source IP, thread ID, and data size.
- For TCP_CONNECT (or TCP_SEND), find TCP_ACCEPT (or TCP_RECV) satisfying (1) its source IP and destination IP are exchanged; (2) its source port and destination port are exchanged.

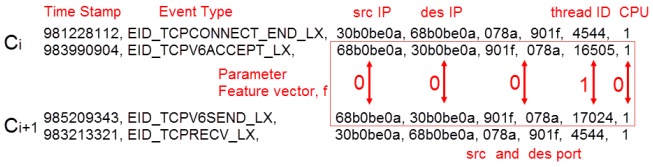

Figure 4. Illustration of parameter feature vector. In this case, the parameter domains include a source IP, a destination IP, a port number pair, a thread ID and a CPU ID.

- For TCP_SHUTDOWN (or TCP_CLOSE), find TCP_RECV satisfying (1) its source IP and destination IP are exchanged; (2) its source port and destination port are exchanged; (3) its data size is zero.

C. Learning Transition Processes of Communication Event Pairs

The rationale behind PInfer's learning approach is that although there can be various request paths, the pairwise relations between communication event pairs is relatively stable. The reason for the stability is that such communication events have to follow the specific protocols and threading models configured in the system (e.g., the TCP protocol). For example, after an external TCP_RECV event is committed, the following communication event pair $\langle$TCP_CONNECT, TCP_ACCEPT$\rangle$ must follow. Therefore, given an event trace corresponding to a series of sequential requests, PInfer learns the transition process of communication event pairs by identifying the frequent event pair-wise transition patterns.

Specifically, we model the transition process of communication event pairs as a Markov process. Given the aligned communication event pairs detected from training sequence, we can estimate the transition probability among communication event pairs statistically, e.g., the 1st order transition probability $p_c(C_i|C_j)$, the 2nd order transition probability $p_c(C_i|C_j \rightarrow C_k)$, and so on. In the Markov process, each communication event pair is considered as a state, and we need to learn the transition probabilities among the states so as to infer the process. Besides the transition processes, the parameter of event also provides us with significant information for inferring request path. In addition, the parameters of the events also provide useful information for inferring request paths. As a matter of fact, the transition process of communication event pairs is generally associated with the change of event parameters. Different from the types of communication event pairs, the values of parameters are numerous and random (e.g., both dynamic allocated port number, the ID of thread, etc). To capture the parameter changes so as to infer the transition process, we propose an effective feature extraction method as illustrated in Fig. 4.

Given adjacent communication event pairs $\{C_i, C_{i+1}\}$, we compare the parameter of C_i's ending event, $\boldsymbol{v}_i^s$ with that

of C_{i+1}'s starting event, $\boldsymbol{v}^b_{i+1}$. If v^c_i and v^b_{i+1} correspond to S domains of system information (i.e., $v = \{v(s)\}_{s=1}^{S}$, we define a binary feature vector $\boldsymbol{f} \in \{0,1\}^S$ as follows to capture the parameter change from v^c_i to v^b_{i+1}) where $f(s)$ represents the change on the s-th domain.

$$f(s) = \begin{cases} 0, & v^s_i(s) = v^b_{i+1}(s), \\ 1, & \text{otherwise.} \end{cases} \tag{4}$$

The feature vector $\boldsymbol{f}$ maps numerous parameter changes into a finite state space. As a result, given learned communication event pairs, we can calculate the conditional probability of parameters given the transition of communication event pairs, denoted as $p_f(\boldsymbol{f}|C_i \to C_{i+1})$.

The transition probability of communication event pairs p_c and the conditional probability of parameters p_f provide us with significant prior knowledge for generating communication paths and the corresponding request processing paths, which will be shown in the next Section.

V. PInfer Inference Scheme

Given the communication process models learned from Section IV, we formulate a generalized graph assignment problem for decoupling interleaving request processing paths on a set of testing data containing concurrent requests, and design an efficient algorithm to solve it.

A. Graph-based Communication Path Generation

For an event sequence that processes K requests, assume we can detect N communication pairs $\{C_i\}_{i=1}^{N}$ and K the starting and ending events for the requests, denoted as $\{C^b_k,\ C^s_k\}_{k=1}^{K}$. We aim to generate K communication paths starting from e^b_k and ending at e^s_k respectively. The most significant challenge in identifying such communication paths is that when the requests are concurrent, their communication paths are interleaved with each other and thus it is non-trivial to decouple them. We propose a graph-based heuristic method to identify the communication paths in a greedy and sequential fashion. In specific, we assign communication pairs to communication paths step by step. At each step, the communication pairs are assigned based on the solution of a generalized assignment problem.

Specifically, given current partially identified communication paths, we first find some communication paths that could possibly be the next pairs in the paths. Given adjacent communication pairs $\{C_i,\ C_{i+1}\}$, where $C_i = \langle e^b_i\ e^s_i \rangle$, and $C_{i+1} = \langle e^b_{i+1}\ e^s_{i+1} \rangle$, there are three matching patterns:

- v^s_i of e^s_i match v^b_{i+1} of e^b_{i+1}, that is C_{i+1} starts after C_i ends.
- v^s_i of e^s_i match v^s_{i+1} of e^s_{i+1}, that is C_{i+1} ends after C_i ends.
- v^b_i of e^b_i match v^b_{i+1} of e^b_{i+1}, that is C_{i+1} starts after C_i starts.

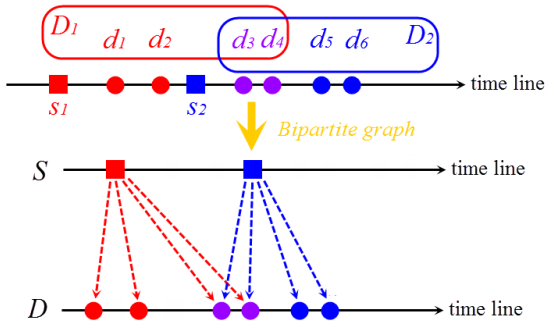

Figure 5. Generalized assignment problem for inferring communication path. The squares are current last pairs of communication paths, and the circles are candidate following pairs. The purple circles represents the candidates shared by two paths.

Those patterns assure the context relevance (e.g., the same TCP connection or thread context) between adjacent communication pairs.

However, expanding processing paths solely based on the matching between consecutive pairs (i.e., 1st-order information) is highly subject to error propagation. This is due to the fact that the ordering of communication pairs recorded by the system can be different from their logical and true ordering. To alleviate the fragility from only the 1st-order information, we propose a high-order (e.g., 2nd-order) path expansion approach as follows. That is, a pair is assigned to a path as long as it can be connected with either C_{i-1} or C_i based on parameter matching. When there are multiple concurrent request paths, different paths may share same next communication pair candidates. In this case, we need to well establish one-to-one assignments between multiple paths and next pair candidates. If the communication path is of length L, we need to repeat the above steps L times, which can be time consuming. In addition, global optimality for all concurrent requests requires enumeration for all possible paths, which may cause scalability issues for a large system with many concurrent requests. On the other hand, expanding communication paths sequentially (i.e., locally optimally) may lead to a cascading of inference errors. If a communication pair is assigned to a wrong path, this error would impact all the following inference along the line.

To achieve a balance among robustness, scalability and error propagation, we propose a locally optimal solution. For the k-th communication path, denote its current last pair and the candidate set of the next pairs as s_k and $\mathcal{D}_k$, respectively. It is possible that $\mathcal{D}_k \cap \mathcal{D}_{k'} \neq \emptyset$, $k \neq k'$. We consider $\mathcal{S} = \{s_k\}_{k=1}^{K}$ as a source set and $\mathcal{D} = \cup_{k=1}^{K} D_k$ as a destination set. These two sets construct a bipartite graph, where the weight of edge connecting the source node $s \in \mathcal{S}$ with the destination node $d \in \mathcal{D}$ is denoted as w_{sd}. Then we construct a bipartite graph over these two sets by connecting each source node $s \in \mathcal{S}$ with each destination node $d \in \mathcal{D}$

with an edge with weight w_{sd}. Taking advantages of the transition process learned before, we can define w_{sd} as where the weight w_{sd} is defined as follows,

$$w_{sd} = p_c(d|s)p_f(f_{sd}|s \to d) \tag{5}$$

when the 1st-order transition probability is applied, or

$$w_{sd} = p_c(d|s' \to s)p_f(f_{sd}|s \to d), \tag{6}$$

when the 2nd-order transition probability is applied.

The inference of the following pairs using the bipartite graph is again an example of a generalized assignment problem [21], [22], which can be addressed by solving the following optimization problem.

$$\begin{aligned} \max_{x_{sd}} \quad & \sum_{s=1}^{|\mathcal{S}|}\sum_{d=1}^{|\mathcal{D}|} w_{sd}x_{sd}, \\ s.t. \quad & x_{sd} \in \{0,1\}, \\ & \sum_{d=1}^{|\mathcal{D}|} x_{sd} = 1, \ s = 1,..,|\mathcal{S}|, \\ & \sum_{s=1}^{|\mathcal{S}|} x_{sd} \leq 1, \ d = 1,..,|\mathcal{D}|. \end{aligned} \tag{7}$$

Similarly to that in (3), the first constraint ensures binary connection indicator. The second constraint ensures that each pair connects exactly one next pair. The third constraint ensures that each candidate connects at most one previous pair. The objective function can be interpreted as to maximize the joint transition probability of communication pairs and their parameters. The difference is that in (7), it does not need to estimate the transformation between the source set and the destination set. The problem in Equation (7) can be solved using the approximate algorithms proposed in [21], [22]. Therefore, the proposed graph-based communication path generation is presented in Algorithm 2.

Algorithm 2 Communication Path Identification Algorithm

Input: Sequence of communication pairs $\{C_i\}_{i=1}^N$, the starting and ending events $\{C_k^b, C_k^e\}_{k=1}^K$, event's transition probability p_e and parameter's conditional probability p_f.
Output: The communication path $\{P_k\}_{k=1}^K$.
for $k = 1 : K$ **do**
  Initialize $P_k = \{C_k^b, C_k^s\}$.
  repeat
    Candidate selection:
    Find a candidate set $\mathcal{D}_k$ for the current pair s_k from the interval between s_k and C_k^e.
    if $\mathcal{D}_k \neq \emptyset$ **then**
      Solve Eq. (7) to obtain the following pairs $\{d_k\}_{k=1}^K$.
      $s_k = d_k$, $P_k = P_k \cup \{s_k\}$.
    end if
  until $\mathcal{D}_k = \emptyset$
end for

B. Request Processing Path Identification

After identifying a communication path, the corresponding request processing path can be generated by filling adjacent communication events in each thread on the communication path with the rest of system kernel events triggered by that thread.

VI. Discussions

A. Computational Complexity

In PInfer, both the communication pair detection and time alignment algorithm (Algorithm 1) and the communication path generation algorithm (Algorithm 2) are graph-based. Specifically, Algorithm 1 solves a constrained least squares problem and a binary programming alternatively. Algorithm 2 solves generalized assignment problems. Both of these two algorithms address NP-hard problems, and thus we solve them approximately.

Algorithm 1 is similar to the ICP algorithm, which selects suitable correspondences for communication events and learns the transitions between them for arbitrary two machines. The difference between our method and traditional ICP has two folds: (1) we consider the causality within communication pairs when learning transformation; (2) we apply local search window for each event, as Fast ICP [23] does. The computational complexity of Algorithm 1 in worst case is $\mathcal{O}(D^2N^2)$, where D is the number of machines and N is the number of inter-communication pairs in the event sequence. When we align the time clocks of machine d and $d+1$ sequentially, $d = 1, ..., D$, and apply the local search window scheme, the computational complexity can be reduced to $\mathcal{O}(DN)$.

Algorithm 2 solves L generalized assignment problems (GAP) for the event sequence of K paths, with maximum length L. We apply the efficient approximate algorithm in [21] to solve each GAP, which solve K knapsack problems [24] repeatedly. Suppose that there are $|\mathcal{D}|$ communication pairs waiting for assignment. For each path, the computational complexity of knapsack problem is $\mathcal{O}(f(|\mathcal{D}|))$. Thus, the computational complexity of Algorithm 2 is $\mathcal{O}(LK(f(|\mathcal{D}|) + |\mathcal{D}|))$.

It should be noted that Algorithm 2 generates communication paths in parallel; after solving GAP, K pairs are assigned to K paths simultaneously. Compared with the method that generates paths sequentially through beam search [25], our method can suppress the cascading effect of error greatly, which will be demonstrated in Section VII.

B. Significance of Domain Knowledge

The event sequences from the target networked system have to always follow the rule of the system. Therefore, a prior domain knowledge of the system plays an important role in understanding the sequences. For example, the event parameter matching is critical for detecting the next communication pairs in a communication path, which heavily

Table III
THE SYSTEM CONFIGURATIONS.

x	**Workload pattern**
1	Overload of CPU (CPU).
2	A certain period of time to sleep (SLEEP).
3	Read the file from disk (IOREAD).
4	Create a new file to disk (IOWRITE).
5	Receive the results of single response record (SELECT1).
6	Do communication and OUTER external system (OUTER).
7	Receive the results of multiple response records (SELECT2).
8	Lock the shared resources during processing (LOCK).
y	**Request processing pattern**
1	Synchronous request.
2	Asynchronous request.
3	Asynchronous request with new thread created.
4	Asynchronous request without return.
5	Asynchronous request with new thread and without return.

relies on the understanding of the event parameters based on domain knowledge, and the feature dimension of $\boldsymbol{f}$ depends on the system. In addition, the parameter matching criteria are determined by the domain knowledge on kernel events.

VII. EVALUATION

A. Implementation

We implemented PInfer as a module of a NEC system performance diagnosis tool (a.k.a. CLUE) [12], which is a kernel event tracing and analysis framework that supports various monitoring platforms such as Linux, HP-UX, ARM Linux, and Windows.

B. Traces

The test traces were collected from a service system of three tiers: a *nginx* [11] event-driven web server, a *Jetty* [10] application server, and a *MySQL* [26] database server. Kernel event traces were collected under multiple workload patterns and request processing patterns. A kernel event data set is presented as a form, "$x - y - z$" where x, y, z are the notation presented in Table III. $x = 1, .., 8$, which represents 8 different workload patterns of system listed in Table. III. $y = 1, .., 5$, which represents 5 request processing patterns listed in Table III. And $z = a, b$, which represents 2 request patterns: (a) sequential request paths and (b) concurrent request paths. In evaluation, we use two sets of trace event sequences for training and testing where each set contains 10-12 requests. The training set contains sequential requests represented as the $x-y-a$ patterns. The event sequences in the testing set can be concurrent. Therefore, the sequences in this set is presented as the $x - y - b$ pattern.

The kernel events in our data set contain the following parameters: the IP of the machine generating the event (**Source IP**); the ID of the thread (or process) generating the event (**Thread ID**); the index of the CPU generating the event (**CPU ID**). For the events related to communication, additional parameter domains are attached: the IP of the target machine (**Destination IP**); the source and the destination port number (**Port**); data size. Except the domain of data size, the dimension of feature vector $\boldsymbol{f}$ is 5 and the number of feature states is $2^5 = 32$.

C. Methods

We present the evaluation results using the comparison of prior work and several design choices of our method. For each one's notation, we use several terms.

- **T**: using a training set
- **1C** and **2C**: the 1st and 2nd order communication pair transition probability respectively
- **F**: parameter conditional probability
- **P**: parallel inference of communication paths, where all paths are updated simultaneously by solving a generalized assignment problem.

The list of compared methods are as follows: The average precision and recall are presented in Table IV and V.

- **NoTrain**: an end-to-end request path profiling method we implemented in the earlier version of *CLUE* [12]. It requires no training, generating communication paths via stitching communication pairs with matched parameters based on domain knowledge.
- **T1C-S**: a method learning the 1st order pair transition probability and inferring path sequentially via beam search [25].
- **T1C-P**: the PInfer method learning the 1st order pair transition probability and inferring paths simultaneously.
- **T2C-P**: the PInfer method learning the 2nd order pair transition probability and inferring paths simultaneously.
- **T2CF-P**: the PInfer method learning the 2nd order pair transition probability and the conditional probability of feature, and then inferring paths simultaneously.

Here we use *precision* and *recall* of inference results to evaluate the performance of various methods. Specifically, we call the events being inferred correctly as positive events. The precision is defined as the proportion of positive events in the inferred request processing paths, and the recall is defined as the proportion of positive events in the actual request processing paths. The precision measures the accuracy of inferred paths while the recall measures the completeness of inferred paths.

D. Results

For each workload pattern, the average inference results of concurrent request processing paths are presented in Tables IV and V. We can find that merely using domain knowledge without a training set cannot deal with concurrent request processing paths accurately; the highly interleaved paths will lead multiple events with matched parameters that appear together in a short time. Stitching matched

Table IV
RESULTS OF AVERAGED PRECISION.

Workload	NoTrain	T1C-S	T1C-P	T2C-P	T2CF-P
CPU	0.6051	0.6547	0.6331	0.6541	0.6780
SLEEP	0.6035	0.6572	0.6423	0.6909	0.6682
IOREAD	0.5559	0.6697	0.6433	0.6742	0.6647
IOWRITE	0.5888	0.6703	0.6349	0.6807	0.6507
SELECT1	0.5082	0.4861	0.5014	0.5463	0.5583
OUTER	0.5743	0.5359	0.6244	0.6105	0.6583
SELECT2	0.5262	0.4750	0.5576	0.5840	0.6016
LOCK	0.5905	0.6563	0.6549	0.6953	0.6815
Average	0.5691	0.6006	0.6115	0.6420	**0.6452**

Table V
RESULTS OF AVERAGED RECALL.

Workload	NoTrain	T1C-S	T1C-P	T2C-P	T2CF-P
CPU	0.6932	0.8511	0.8364	0.8364	0.8614
SLEEP	0.7557	0.8875	0.8693	0.8489	0.8682
IOREAD	0.6806	0.8290	0.8426	0.8743	0.8801
IOWRITE	0.6782	0.8602	0.8614	0.8559	0.8636
SELECT1	0.7063	0.7916	0.7925	0.7854	0.8008
OUTER	0.7160	0.8751	0.8338	0.8528	0.8838
SELECT2	0.7075	0.8097	0.7598	0.7758	0.7933
LOCK	0.7432	0.8398	0.8034	0.8239	0.8545
Average	0.7101	0.8430	0.8249	0.8317	**0.8507**

communication pairs only based on "closest" criterion and generating request processing paths accordingly introduced many errors as well. The data show that by introducing the event transition probability into the inference, the result has been improved significantly. However, if the inference is sequential, the inference error will be cascading — the error of the inference for a previous path will affect the errors in the following paths. By applying the 2nd order transition probability and the parallel inference method, the robustness and the accuracy of the inference are further improved. Additionally, taking the parameter conditional probability into consideration, the inference accuracy is slightly better. It means that the parameter conditional probability further provides us with useful information. PInfer achieves on average 65% precision and 85% recall.

The experiment results demonstrate that as more prior knowledge is introduced in the inference, the result becomes better. As a result, our method achieves superior performance in all cases. The high precision means that there are few false positive errors in our result while the high recall means that our inference result covers most of events for real paths. To further verify the advantage of our method, Table VI visualizes the inference results of various methods for a typical request processing path. The "–" indicates the false negative events (belonging to the path but not inferred) while the red events indicate the false positive events (not belonging to the path but inferred). Compared with the prior work, our method attains a path with fewer inference errors indeed.

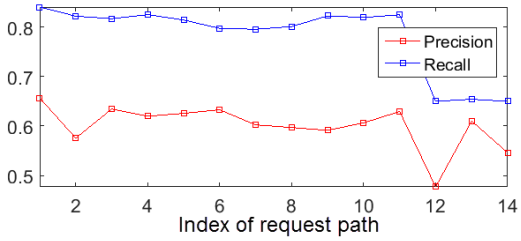

Figure 6. The precision and recall of 14 inferred paths w.r.t. the index of request.

E. Discussions

Although our method achieves encouraging results for inferring request processing paths, it has some limitations. First, our method requires a training set consisting of sequential request processing paths to learn the transition process of communication paths.

Second, although our path generation algorithm suppresses the cascading effects of inference errors, it does not essentially solve this problem. Because our algorithm adds communication pairs into the inference of the paths step-by-step, the final inferred paths are local optimal being affected by the propagation of inference errors.

We further give data in Figure 6 to illustrate this phenomenon of error propagation. For a sequence having 14 concurrent requests, their inferred paths are sorted according to their starting time. In all methods, the inferred results of the following requests are generally worse than those of the previous ones, which means that the errors of previous inference results influence the following inference.

VIII. CONCLUSION

In this paper, we present PInfer, a novel method to infer concurrent request processing paths with system event traces. The method combines domain knowledge and the learned behavior model from training data, and enables inferring interleaving request processing paths on asynchronous and configurable multi-threading service systems. We implemented and tested it on a wide range of request processing patterns on a multi-tiered service system, and the results showed the effectiveness and practicality of PInfer.

REFERENCES

[1] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," in *Proceedings of the 2005 Ottawa Linux Symposium (OLS)*, 2005.

[2] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX'04*.

[3] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu, "Fay: extensible distributed tracing from kernels to clusters," in *SOSP '11*, 2011.

Table VI
COMPARISON OF INFERENCE RESULTS

Ground Truth	NoTrain	T1C-S	T1C-P	T2C-P	T2CF-P
TCPRECV	TCPRECV	TCPRECV	TCPRECV	TCPRECV	TCPRECV
TCPCONNECT_END	TCPCONNECT_END	TCPSEND	TCPCONNECT_END	TCPCONNECT_END	TCPCONNECT_END
TCPV6ACCEPT	TCPV6ACCEPT	TCPV6RECV	TCPV6ACCEPT	TCPV6ACCEPT	TCPV6ACCEPT
TCPSEND	–	TCPCONNECT_END	TCPSEND	TCPSEND	TCPSEND
TCPV6RECV	–	TCPV6ACCEPT	TCPV6RECV	TCPV6RECV	TCPV6RECV
PIPEWRITE	–	TCPSEND	TCPV6SEND	TCPV6SEND	PIPEWRITE
PIPEREAD	–	TCPV6RECV	TCPRECV	TCPRECV	PIPEREAD
SETRQ	TCPV6SHUTDOWN	PIPEWRITE	SETRQ	PIPEWRITE	SETRQ
PRESUME	TCPRECV	PIPEREAD	PRESUME	PIPEREAD	PRESUME
TCPCLOSE	TCPCLOSE	SETRQ	–	–	PIPEWRITE
TCPV6RECV	TCPV6RECV	PRESUME	–	–	PIPEREAD
SETRQ	–	PIPEWRITE	–	TCPCLOSE	SETRQ
PRESUME	–	PIPEREAD	–	TCPV6RECV	PRESUME
TCPSEND	TCPSEND	SETRQ	TCPSEND	–	TCPCLOSE
		PRESUME		–	TCPV6RECV
		TCPCLOSE		TCPSEND	TCPSEND
		TCPV6RECV			
		–			
		–			
		TCPSEND			

[4] T. Marian, H. Weatherspoon, K.-S. Lee, and A. Sagar, "Fmeter: Extracting indexable low-level system signatures by counting kernel function calls," in *Middleware*, ser. Lecture Notes in Computer Science, P. Narasimhan and P. Triantafillou, Eds., vol. 7662. Springer, 2012, pp. 81–100.

[5] R. J. Moore, "A universal dynamic trace for linux and other operating systems," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 297–308.

[6] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vpath: Precise discovery of request processing paths from black-box observations of thread and network activities," in *USENIX'09*.

[7] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *Linux Symposium*, June 2006.

[8] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu, "Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces," in *SIGMETRICS '14*.

[9] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "Perfscope: Practical online server performance bug inference in production cloud computing infrastructures," in *SOCC '14*.

[10] Jetty, "http://www.eclipse.org/jetty/."

[11] N. open source web server, "https://www.nginx.com."

[12] H. Zhang, J. Rhee, N. Arora, S. Gamage, G. Jiang, K. Yoshihira, and D. Xu, "CLUE: system trace analytics for cloud service performance diagnosis," in *NOMS 2014*.

[13] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *SOSP '03*, 2003.

[14] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "Wap5: black-box performance debugging for wide-area systems," in *WWW '06*.

[15] E. Koskinen and J. Jannotti, "Borderpatrol: isolating events for black-box tracing," in *Eurosys '08*.

[16] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, and Z. Zhang, "Precise, scalable, and online request tracing for multi-tier services of black boxes," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2011.

[17] X. Meng, G. Jiang, H. Zhang, H. Chen, and K. Yoshihira, "Automatic profiling of network event sequences: Algorithm and applications," in *INFOCOM '08*.

[18] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *SIGCOMM '15*.

[19] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Robotics-DL tentative*. International Society for Optics and Photonics, 1992, pp. 586–606.

[20] D. Chetverikov, D. Svirko, D. Stepanov, and P. Krsek, "The trimmed iterative closest point algorithm," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 3. IEEE, 2002, pp. 545–548.

[21] R. Cohen, L. Katzir, and D. Raz, "An efficient approximation for the generalized assignment problem," *Information Processing Letters*, vol. 100, no. 4, pp. 162–166, 2006.

[22] L. Fleischer, M. X. Goemans, V. S. Mirrokni, and M. Sviridenko, "Tight approximation algorithms for maximum general assignment problems," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, 2006.

[23] T. Jost and H. Hügli, "Fast icp algorithms for shape registration," in *Pattern Recognition*. Springer, 2002, pp. 91–99.

[24] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[25] P. Norvig, *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann, 1992.

[26] MySQL, "https://www.mysql.com/."