

# Precise, Scalable, and Online Request Tracing for Multi-tier Services of Black Boxes

Bo Sang, Jianfeng Zhan, Gang Lu, Zhihong Zhang, Haining Wang, Dongyan Xu, Lei Wang, Dan Meng

**Abstract**—As more and more multi-tier services are developed from commercial off-the-shelf components or heterogeneous middleware without source code available, both developers and administrators need a request tracing tool to (1) exactly know how a user request of interest travels through services of black boxes; (2) obtain macro-level user request behaviors of services without manually analyzing massive logs. This need is further exacerbated by IT system “agility”, which mandates tracing tools to provide on-line performance data since off-line approaches cannot reflect system changes in real time. Moreover, considering the large scales of the deployed services, a pragmatic tracing approach should be scalable in terms of the cost in collecting and analyzing logs. In this paper, we introduce a precise, scalable, and online request tracing tool for multi-tier services of black boxes. Our contributions are three-fold. First, we propose a precise request tracing algorithm for multi-tier services of black boxes, which only uses application-independent knowledge. Second, we present a micro-level abstraction, *component activity graph*, to represent causal paths of each request. On the basis of this abstraction, we propose *dominated causal path patterns* to represent repeatedly executed causal paths that account for significant fractions. We further present a derived performance characteristic of causal path patterns, *latency percentages of components*, to enable debugging performance-in-the-large. Third, we develop two mechanisms: *tracing on demand* and *sampling* to significantly increase system scalability. We implement a prototype of the proposed system, called *PreciseTracer*, and release it as open source code. In comparison with WAP5—a black-box tracing approach, *PreciseTracer* achieves higher tracing accuracy and faster response time. Our experimental results also show *PreciseTracer* has low overhead, and still achieves high tracing accuracy even if an aggressive sampling policy is adopted, indicating that *PreciseTracer* is a promising tracing tool for large-scale production systems.

**Index Terms**—Multi-tier service, black boxes, precise request tracing, micro and macro-level abstractions, online analysis, performance debugging, scalability.

## 1 INTRODUCTION

More and more multi-tier services or cloud applications [36] are deployed on data centers. In order to pinpoint performance bottlenecks, shoot misconfigurations, or even accurately tune power-saving management policies [32], both developers and administrators need a tracing tool to (1) know how exactly a user request or job of interest travels through services or job execution frameworks, e.g., a MapReduce runtime system [36] if necessary; (2) obtain macro-level user behaviors without analyzing massive logs. The need for online tracing is further exacerbated by IT system agility [22]. For example, in data centers, users increasingly require service support under peak loads that are an order of magnitude greater than those loads in a normal steady state [30]. To deal with fluctuated workloads, resources are often dynamically provisioned and service instances are adjusted accordingly. In this context, the *on-line* performance information is imperative, which only can be obtained with an online tracing tool because off-line tracing approaches cannot reflect system changes in real time. Moreover, understanding online behavior

is also important under normal operating conditions for online scheduling and dispatching decisions [7] for either multi-tier web services [30] or high throughput computing (HTC) or many task computing workloads (MTC) [37] [38] in consolidated cloud computing systems.

This paper focuses on online request tracing of multi-tier services of *black boxes*, since more and more multi-tier services are developed from commercial off-the-shelf components or heterogeneous middleware *without source code available*. Different from profiling [35] that is measurement of a statistical summary of the behavior of a system, tracing is a measurement of a stream of events of the behavior of a system [39]. In this paper, when we refer to *precise request tracing*, it implies the accurate tracking of how a user request of interest travels through services.

Precise request tracing for multi-tier services of black boxes is challenging in many aspects. First of all, we can not access the source code of multi-tier services of black boxes, so it is difficult to understand the contexts of requests or even network protocols [5]. Second, a precise request tracing tool is needed to exactly track a user request of interest if necessary, with the focus on *performance-in-the-large* [4]. However, services are often deployed within a large-scale data center, and a precise request tracing system will inevitably produce massive logs, and hence a pragmatic tracing tool should be scalable with respect to log collection and analysis [27].

---

• Bo Sang, Jianfeng Zhan, Gang lu, Zhihong Zhang, Lei Wang, and Dan Meng are with Institute of Computing Technology, Chinese Academy of Sciences. Haining Wang is with the Department of Computer Science, College of William and Mary. Dongyan Xu is with the Department of Computer Science, Purdue University. Jianfeng Zhan is the contact person.

Moreover, macro-level abstractions are required to facilitate debugging performance-in-the-large. Finally, those tools should not degrade the performance of multi-tier services.

The most straightforward and accurate way [27] [11] [1] [10] [15] [12] to correlate message streams is to leverage application-specific knowledge and explicitly declare causal relationships among events of different components. Its disadvantage is that users must obtain and modify the source code of target applications or middleware, or it even requires that users have in-depth knowledge of target applications or instrumented middleware. Thus, this approach cannot be used for services of black boxes. Without knowledge of the source code, several previous approaches [4] [3] [7] either use imprecision of probabilistic correlation methods to infer average response time of components, or rely upon the knowledge of protocols to isolate events or requests for precise request tracing [5]. In [22], a precise but unscalable request tracing tool, called *vPath*, is proposed for services of black boxes. Because of its limitation in the implementation, the tracing mechanism in *vPath* cannot be enabled or disabled on demand without interrupting services. Thus, it has to continuously collect and analyze logs, which results in unacceptably high cost. Moreover, state-of-the-art precise request tracing approaches of black boxes [5] [22] fail to propose abstractions for representing macro-level user request behaviors, and instead depend on users' manual interpretation of massive logs in debugging performance-in-the-large. Besides, they [5] [20] [22] are offline schemes.

In this paper, we present a precise and scalable request tracing tool for online analysis of multi-tier services of black-boxes. Our tool collects activity logs of multi-tier services through kernel instrumentation, which can be enabled or disabled on demand. Through tolerating log losses, our system supports *sampling* or *tracing on demand*, which significantly decreases the collected and analyzed logs and improves the system scalability. After reconstructing those activity logs into *causal paths*, each of which is a sequence of activities with causal relations caused by an individual request, we classify those causal paths into different *causal patterns*, which represent repeatedly executed causal paths. Then, we present a macro-level abstraction, *dominated causal path patterns*, to represent causal path patterns that account for significant fractions in terms of the number of causal paths. On a basis of this, we propose a derived performance characteristic of major causal path patterns, *latency percentages of components*, to enable debugging performance-in-the-large.

We implement a prototype of the proposed system, called *PreciseTracer*, and release it as open source code<sup>1</sup>. We perform extensive experiments on 3-tier platforms. Our experimental results show that: (1) with respect

to WAP5—a black-box tracing approach [3], *PreciseTracer* achieves higher tracing accuracy and faster response time. (2) *PreciseTracer* has low overhead, and still achieves high tracing accuracy even if an aggressive sampling policy is adopted. (3) Our derived performance characteristic of causal path patterns, latency percentages of components, enables debugging performance-in-the-large.

Summarily, we make the following contributions in this paper.

- 1) We design a precise tracing algorithm to deduce causal paths of requests from interaction activities of components of black boxes. Our algorithm only uses application-independent knowledge.
- 2) We present two abstractions, component activity graphs and dominated causal path patterns, to represent individual causal path for each request and macro-level user request behaviors, respectively. Based on these abstractions, we propose a derived performance characteristic, latency percentages of components, to enable debugging performance-in-the-large.
- 3) We present two mechanisms: tracing on demand and sampling, to improve system scalability.

The remainder of the paper is organized as follows. Section 2 formulates the problem. Section 3 describes the design of *PreciseTracer*. Section 4 details the implementation of *PreciseTracer*. Section 5 evaluates the performance of *PreciseTracer* with respect to WAP5. Section 6 summarizes related work. Section 7 draws a conclusion.

## 2 SYSTEM MODEL AND PROBLEM STATEMENT

### 2.1 System model

Our target environments are data centers deployed with multi-tier services. There are two types of nodes in this environment: *service nodes* and *analysis nodes*. Service nodes are the ones on which multi-tier services are deployed, while most components of the tracing tool are deployed on analysis nodes.

The application assumptions are as follows:

- We treat each component in a multi-tier service as a black box, since we cannot obtain the application or middleware source code, neither deploy the instrumented middleware, nor have the knowledge of high-level protocols used by services, like http etc.
- We presume that a single execution entity (a process or a kernel thread) of each component can only serve one request in a certain period. For serving each individual request, execution entities of components cooperate through sending or receiving messages via a reliable communication protocol, like TCP. An individual request is tracked by monitoring a series of activities, which have causal relations for tracing requests.

Though not all multi-tier services fall in the scope of our target applications, fortunately many popular

1. The source code is downloadable from <http://www.ncic.ac.cn/~zjf>.

services satisfy our assumed scenarios. For example, our method can be used to analyze concurrent servers following nine design patterns introduced in [6] (Chapter 27), including iteration, concurrent process, concurrent thread, preforking and prethreading models.

## 2.2 Problem statement

As shown in Fig. 1, a request causes a series of *interaction activities* in the operating system kernel, e.g. sending or receiving messages. Those activities occur under specific contexts (*processes* or *kernel threads*) of different components. We record an activity of sending a message as  $S_{i,j}^i$ , which indicates a process  $i$  sends a message to a process  $j$ . We record an activity of receiving a message as  $R_{i,j}^j$ , which indicates a process  $j$  receives a message from a process  $i$ .

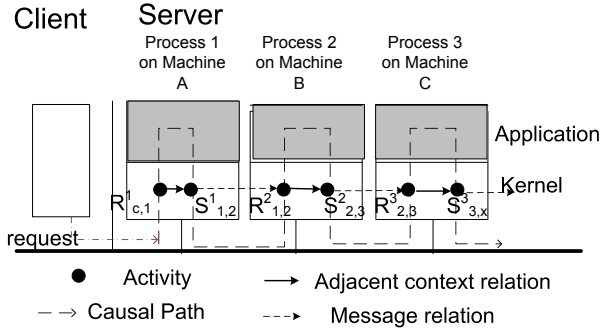


Fig. 1. Activities with causal relations in the kernel.

When an individual request is serviced, a series of activities having causal relations or happened-before relationships as defined by Lamport [8] constitute a *causal path*, e.g., in Fig. 1, the activity sequence  $\{R_{c,1}^1, S_{1,2}^1, R_{2,2}^2, S_{2,3}^2, R_{2,3}^3, S_{3,x}^3\}$  constitutes a causal path. For each individual request, there is a causal path.

Given the scenario above, the problems we tackle in this paper can be described as follows: with the logs of instrumented communication activities, how can we obtain individual causal paths that precisely correlate activities of components for each request. How to obtain dominated causal path patterns, which represent repeatedly executed causal paths, and their on-line or off-line performance data?

Based on the data provided by PreciseTracer, developers and administrators can accurately track how a user request of interest travels through services, debug performance problems of a multi-tier service, and provide online performance data of services for the feedback controller in the runtime power management system [32].

## 3 PRECISETRACER DESIGN

In this section, we first present the architecture of PreciseTracer and its abstractions. Then, we detail the tracing algorithm and the mechanisms for improving the system scalability.

### 3.1 PreciseTracer Architecture

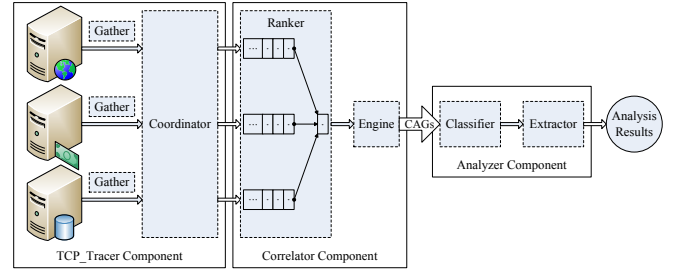


Fig. 2. PreciseTracer Architecture.

PreciseTracer is flexible, and administrators can configure it according to their requirements. PreciseTracer can work under either an offline mode or an online mode. Under an offline mode, PreciseTracer only analyzes logs after collecting them for a relatively long period of time. Under an online mode, PreciseTracer collects, and analyzes logs in a simultaneous manner, providing administrators with real-time performance information.

Fig. 2 shows the architecture of PreciseTracer, which consists of three major components: *TCP\_Tracer*, *Correlator*, and *Analyzer*. *TCP\_Tracer* records interaction activities of interest for the components of target applications, and output those logs to *Correlator*. *Correlator* is responsible for correlating those activity logs of different components into causal paths. Finally, based on the causal paths produced by *Correlator*, *Analyzer* extracts useful information, and reports analysis results.

*TCP\_Tracer* includes two modules: *Gather* and *Coordinator*. Deployed on each service node, *Gather* is responsible for collecting logs of services deployed on the same node. *Coordinator*, which is deployed on an analysis node, is in charge of controlling *Gather* on each service node. *Coordinator* configures and synchronizes *Gather* on each service node to send logs to *Correlator*.

*Gather* independently observes interaction activities of components of black boxes on each node. Concentrating on the major activities, *Gather* only cares about when serving a request starts, finishes, and when components receive or send messages *within the confine of a data center*. Of course, we can observe more activities if the overhead is acceptable. In this paper, our observed activity types include: *BEGIN*, *END*, *SEND*, and *RECEIVE*. *SEND* and *RECEIVE* activities are the ones of sending and receiving messages. A *BEGIN* activity marks the start of serving a new request, while an *END* activity marks the end of serving a request. For each activity, *Gather* records five attributes: (*activity type*, *timestamp*, *context identifier*, *message identifier*), and *message size*. For each activity, we use 4-tuple (*hostname*, *program name*, *process ID*, *thread ID*) to describe its context identifier, and use 5-tuple (*IP of sender*, *port of sender*, *IP of receiver*, *port of receiver*, *message size*) to describe its message identifier.

*Correlator* includes two major modules: *Ranker* and *Engine*. *Ranker* is responsible for choosing candidate

activities for composing causal paths. Engine constructs causal paths from the outputs of Ranker, and then reports causal paths.

Analyzer includes two major modules: *Classifier* and *Extractor*. Classifier is responsible for classifying causal paths into different patterns, while Extractor provides analysis results of causal path patterns.

### 3.2 Abstractions

Formally, we propose a *directed acyclic graph*  $G(V, E)$  to represent causal paths, where vertices  $V$  are activities of components and edges  $E$  represent causal relations between activities. We define this abstraction as *component activity graph* (CAG). For each individual request, a corresponding CAG represents all activities with causal relations in the life cycle of serving the request.

CAGs include two types of relations: an *adjacent context relation* and a *message relation*. We formally define the two relations based on the happened-before relation [8], which is denoted as  $\rightarrow$ , as follows:

*An adjacent context relation.* Caused by the same request  $r$ ,  $x$  and  $y$  are activities observed in the same context  $c$  (a process or a kernel thread), and  $x \rightarrow y$  holds true. If no activity  $z$ , which satisfies the relations  $x \rightarrow z$  and  $z \rightarrow y$ , is observed in the same context, then an adjacent context relation exists between  $x$  and  $y$ , denoted as  $x \rightarrow_c y$ . So, the adjacent context relation  $x \rightarrow_c y$  means that  $x$  has happened right before  $y$  in the same execution entity.

*A message relation.* For serving a request  $r$ , if  $x$  is a *SEND* activity, which sends a message  $m$ , and  $y$  is a *RECEIVE* activity, which receives the same message  $m$ , then a message relation exists between  $x$  and  $y$ , denoted as  $x \rightarrow_m y$ . So, the message relation  $x \rightarrow_m y$  means that  $x$ , which sends a message, has happened right before  $y$ , which receives a message, in two different execution entities.

If there is an edge from activity  $x$  to activity  $y$  in a CAG, of which  $x \rightarrow_c y$  or  $x \rightarrow_m y$  holds true, then  $x$  is the parent of  $y$ .

In a CAG, each activity vertex must satisfy the following property: each activity vertex has no more than two parents, and only a *RECEIVE* activity vertex could have two parents, with which one parent has an adjacent context relation and the other has a message relation. This has two-fold reasons: first, an adjacent context relation is used to describe activities in the same process or thread caused by the same request, so an activity at most has a parent activity that is adjacent to and ahead of it on the time line; second, for a message relation, *SEND* and *RECEIVE* activities always come in pairs.

For an individual request, it is clear that correlating a causal path is the course of building a CAG with interaction activities as the input. Fig. 3 shows an example of an individual CAG.

A single causal path can help administrators get micro-level user request behavior of services. For example, administrators can detect transient failures of

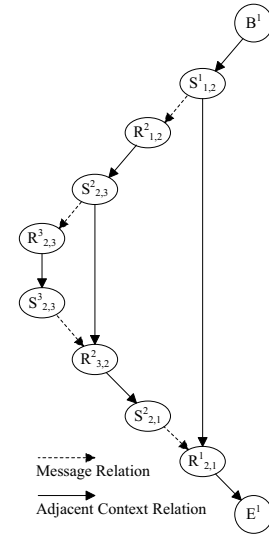


Fig. 3. An example of an individual CAG. The notations are defined in Section 2.

nodes if some causal paths show abnormal information. However, causal paths cannot be directly utilized to represent macro-level performance signature data of services for two reasons. First, there are massive causal paths. An individual causal path only reflects how a request is served by services. Considering disturbance in environments, we cannot take an individual causal path as a service's performance signature data. Second, different types of requests would produce causal paths with different features. Thus, we propose a *macro-level abstraction* to represent performance signature data of multi-tier services.

We propose to classify causal paths into different causal path patterns according to the shapes of CAGs. On the basis of this abstraction, we further consider which ones are dominated causal path patterns according to their fractions in terms of their path numbers, respectively. Two CAGs will be classified into the same causal path pattern when they meet the following criteria:

- 1) There are the same number of activities in the two CAGs.
- 2) Two matching activities with the same order in the two CAGs have the same attribution in terms of (*activity type, program name*).

In a CAG, for each activity, we define its order based on the following rules:

**Rule 1:** If  $x \rightarrow_c y$  or  $x \rightarrow_m y$ , then  $x \prec y$ ;

**Rule 2:** If  $x \rightarrow_c y$  and  $x \rightarrow_m z$  and there is no relation between  $y$  and  $z$ , then  $x \prec z \prec y$ ;

**Rule 3:** If  $y \rightarrow_c x$  and  $z \rightarrow_m x$  and there is no relation between  $y$  and  $z$ , then  $y \prec z \prec x$ .

Rule 1 is obvious. Rule 2 and Rule 3 can be derived from our presumption in Section 2.1. In Section 2.1, we presume that a single execution entity (a process or a

kernel thread) of each component can only serve one request in a certain period, which implies a blocking communication mode. For example, in Rule 2, for  $x \rightarrow_m z$  and  $x \rightarrow_c y$ ,  $x$  must be a SEND activity, and  $z$  must be a RECEIVE activity. As an adjacent activity of  $x$ ,  $y$  must happen after  $z$ , since the single execution entity under which  $x$  happens only triggers  $y$  after the return from the call to the execution entity under which  $z$  happens.

CAGs include rich performance data of services, because a CAG indicates how a client request is served by each component of a service. For example, if administrators want to pinpoint the performance bottleneck of a service, they need to obtain the service time consumed on each component in serving requests. According to a CAG, we can compute the latencies of components in serving an individual request. For example, for the request in Fig. 1, the latency of *process 2* is  $(t(S_{2,3}) - t(R_{1,2}))$ , and the interaction latency from *process 1* to *process 2* is  $(t(R_{1,2}) - t(S_{1,2}))$ , where  $t$  is the local timestamp of each activity. The latency of *process 2* is accurate, since all timestamps are from the same node. The interaction latency from *process 1* to *process 2* is inaccurate, since we do not remedy the clock skew between two nodes. For a request, the *server-side latency* can be defined as the time difference between the time stamp of BEGIN activity and that of END activity in its corresponding causal path. For each tier, its role in serving a request can be measured in terms of *the latency percentage of each tier*, which is the ratio of the accumulated latencies of the tier to the server-side latency. The latency percentage of the interaction between two tiers can be defined as the ratio of the accumulated latencies between two tiers to the server-side latency. We adopt the terminology of *latency percentages of components* to describe both latency percentages of each tier and latency percentage of interactions.

After the classification, we can compute the average performance data about causal path patterns, i.e. latency percentage of each tier and latency percentage of interactions, on a basis of which we can further detect performance problems of multi-tier service or provide online performance data for the feedback controller that aims to save cluster power consumption. Our experiments in Section 5.3 and our work in power management [32] demonstrate the effectiveness of this approach in debugging performance-in-the-large and saving cluster power consumption, respectively.

### 3.3 Tracing algorithm

Before we proceed to introduce the algorithm of Ranker, we explain how Engine stores incomplete CAGs. In the course of building CAGs, all incomplete CAGs are indexed with two index map data structures. An *index map* maps a key to a value, and supports basic operations, like search, insertion, and deletion. One index map, named *mmap*, is used to match message relations, and the other one, named *cmap*, is used to match adjacent

context relations. For *mmap*, the key is the message identifier of an activity, and the value of *mmap* is an unmatched SEND activity with the same message identifier. The key in *cmap* is the context identifier of an activity, and the value of *cmap* is the latest activity with the same context identifier.

In the following, Section 3.3.1 explains how to choose candidate activities in constructing CAGs in our algorithm; Section 3.3.2 introduces how to construct CAGs; and Section 3.3.3 describes how to handle disturbances.

#### 3.3.1 Selection of candidate activities for composing CAGs

For each service node, we choose the minimal local timestamp of activities as the initial time. We set a *sliding time window* for processing the activity stream. Activities, logged on different nodes, will be fetched into the buffer of Ranker if their timestamps are within the sliding time window. Section 3.3.3 will present how to deal with clock skews in distributed systems.

Ranker puts each activity into several different queues according to the IP address of its context identifier. Naturally, activities in the same queue are sorted according to the same local clock, so Ranker only needs to compare head activities of each queue, and selects candidate activities for composing CAGs based on the following rules:

**Rule 4:** If a head activity  $A$  in a queue has the RECEIVE type and Ranker has found an activity  $X$  in the *mmap*, of which  $X \rightarrow_m A$  holds true, then  $A$  is the candidate.

If a key is the message identifier of an activity  $A$  and the value of the *mmap* points to a SEND activity  $X$  with the same message identifier, we can say  $X \rightarrow_m A$ .

Rule 4 ensures that when a SEND activity has become a candidate and been delivered to Engine, the RECEIVE activity having message relation with it will also become a candidate once it becomes a head activity in its queue.

**Rule 5:** If no head activity is qualified with Rule 1, then Ranker compares the type of head activities in each queue according to the priority of  $BEGIN \prec SEND \prec END \prec RECEIVE \prec MAX$ . The head activity with the lower priority is the candidate.

Rule 5 ensures that a SEND activity  $X$  always becomes a candidate earlier than a RECEIVE activity  $A$ , if  $X \rightarrow_m A$  holds true.

After a candidate activity is chosen, it will be popped out from its queue and delivered to Engine, and Engine matches the candidate with an incomplete CAG. Then, the element next to the popped candidate will become a new head activity in that queue. At the same time, Ranker will update the new minimal timestamp in the sliding time window, and fetch new qualified activities into the buffer of Ranker.

#### 3.3.2 Constructing CAG

Engine fetches a candidate, outputted by Ranker, and matches it with an incomplete CAG. In Appendix A, the

pseudo code illustrates the correlation algorithm. In line 1, Engine iteratively fetches a candidate activity *current* by calling function *rank()* of Ranker. From lines 2-37, Engine parses, and handles activity *current* according to its activity type. Lines 3-11 handle BEGIN and END activities. For a BEGIN activity, a new CAG is created. For an END activity, the construction of its matched CAG is completed.

Lines 12-37 handle SEND and RECEIVE activities. Activities are inherently asymmetric between a sender and a receiver because of their underlying buffer sizes and delivery mechanisms. Thus, a match between SEND and RECEIVE activities is not always one-to-one, but *n-to-n* relations. Fig. 4 shows a case in which a sender consecutively sends a message in two parts and a receiver receives messages in three parts. Our algorithm correlates and merges these activities according to the message sizes in the message identifiers. If some RECEIVE activities are lost, the received message size will be less than the sent message size, but this would not prevent the algorithm from constructing a CAG.

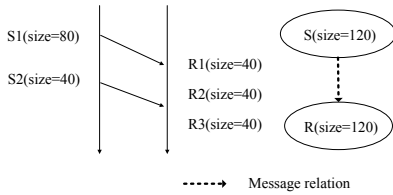


Fig. 4. Merging multiple SEND and RECEIVE activities.

It is possible that an activity is wrongly correlated to two causal paths, because of reusing threads in some concurrent programming paradigms. For example, in a thread-pool implementation, one thread may serve one request at a time; when the work is done, the thread is recycled into the thread pool. Lines 31-33 check if two parents are in the same CAG. If the check returns true, Engine will add an edge of context relation.

### 3.3.3 Disturbance tolerance

In an environment without disturbance, our algorithm can produce correct causal paths. However, in practice, there are many disturbances. In the rest of this subsection, we consider how to resolve noise activities disturbance, concurrency disturbance, and clock skew disturbance.

**Noise activities disturbance.** Noise activities are caused by other applications coexisting with the target service on the same nodes. Their activities through the kernel’s TCP stack will also be logged and gathered by our tool. Ranker handles noise activities in two ways: 1) it filters noise activities according to their attributes, including program name, IP and port; and 2) If activities cannot be filtered with the attributes, the ranker checks them with *is\_noise()* function. If true, the ranker will discard them. The pseudo code of *is\_noise()* function can be found at Appendix A.

**Concurrency disturbance.** The second disturbance is called *concurrency disturbance*, which only exists in multi-processor nodes. Fig. 5-a illustrates a possible scenario, of which two concurrent requests are concurrently served by two multi-processor nodes and four activities are observed.  $S_{1,2}^{1,1}$  means a SEND activity produced on the CPU1 of Node1, and  $R_{1,2}^{2,0}$  is its matched RECEIVE activity produced on the CPU0 of Node2. When these four activities are fetched into the buffer of Ranker, they are put into two queues as shown in Figure. 5a. The head activities of both two queues are RECEIVE activities, and hence they block the matched SEND activities of each other. This case is detected according to two conditions: 1) both head activities of two queues are RECEIVE activities. 2) SEND and RECEIVE activities in two queues are matching with each other, respectively. Ranker handles this case by swapping the head activity and its following activity in the first queue. Figure. 5b illustrates our solution.

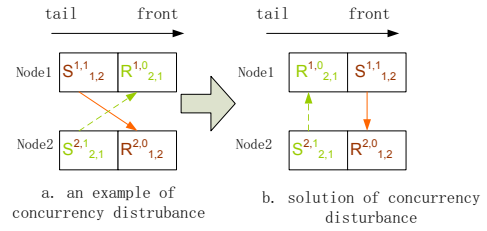


Fig. 5. An example of concurrency disturbance.

**Clock skew disturbance.** As explained in Section 3.3.1, activities will be fetched into the buffer of Ranker according to their local timestamp. However, due to clock skew, RECEIVE activities might be fetched into the buffer before their corresponding SEND activities. We take a simple solution to resolve this issue. Comparing head activities of each queue, we record the timestamp of the first activity from each node. Then, we compute the approximate clock skew between two nodes. Based on the approximate clock skew, we remedy the timestamp of activities on the node with the larger clock skew, and hence we can prevent the scenario mentioned above from happening.

## 3.4 Improving system scalability

We use two mechanisms, tracing on demand and sampling, to improve the system scalability.

### 3.4.1 Tracing on demand

The instrumentation mechanism of PreciseTracer depends on open source software named *SystemTap* [33], which extends the capabilities of *Kprobe* [31]—a tracing tool on a single Linux node. Using *SystemTap*, we have written the *LOG\_TRACE* module, which is a part of Gather. *LOG\_TRACE* obtains context information of processes and threads from the operating system, and further inserts probe points into *tcp\_sendmsg()* and

`tcp_recvmsg()` functions of the kernel communication stack to log sending or receiving activities.

Deployed on each node, Gather receives commands from Coordinator. When PreciseTracer is enabled or disabled on user demand, Coordinator will synchronize each Gather to dynamically load or unload the kernel module LOG\_TRACE, which is supported by the Linux OS. The instrumentation mode of PreciseTracer can be set as *continuous collection*, *tracing on demand*, or *periodical sampling*. When administrators detect the running states of services are abnormal, they can choose the mode of tracing on demand, in which PreciseTracer starts tracing requests according to the commands of administrators. When administrators have pinpointed problems, they can stop tracing requests. When PreciseTracer is set as the mode of periodical sampling, it will be enabled and disabled alternately, which reduces the overhead of PreciseTracer and improves the system scalability.

### 3.4.2 Sampling

Sampling is a straightforward solution to reduce the amount of logs produced by tracing requests of multi-tier services. However, it is not a trivial issue to support sampling in precise request tracing approaches. First, the tracing mechanism should be flexible enough to be enabled or disabled on demand. Second, the tracing algorithm must tolerate log losses. In the following, we discuss how to tolerate losses of activities and consider three different cases:

- **Case 1:** Lost BEGIN and END activities;
- **Case 2:** Lost RECEIVE activities;
- **Case 3:** Lost SEND activities.

It is difficult to handle Case 1. Each CAG needs a BEGIN activity and an END activity to identify its begin and end. Fortunately, losses of BEGIN and END activities only affect the construction of their affiliated CAG, and have no influence on other CAGs whose BEGIN and END activities have been identified.

About Case 2, due to the underlying delivery mechanism, a receiver will receive a message in several parts, which is mentioned in Section 3.3.2. The situation that all parts of a message fail to be collected seldom happens. Therefore, in Case 2, the *received message size* will be less than its corresponding *sent message size*, but this wouldn't prevent us from constructing a CAG.

For Case 3, Fig. 6 shows a scenario of lost SEND activities when candidate activities are in queues of Ranker. Activities from the same node are put into a queue and ordered based on their local timestamps. We hereby utilize (*activity type*, *context identifier*, *message identifier*, *message size*) to identify an activity. In Fig. 6, (SEND, context\_4, message\_3, 60) is related to (RECEIVE, context\_7, message\_3, 60), while (SEND, context\_6, message\_3, 40) is related to (RECEIVE, context\_9, message\_3, 40) and they share the same message identifier. Ranker would pick candidate activities. However, if it fails to collect the activity of (SEND, context\_4, message\_3, 60), the activity

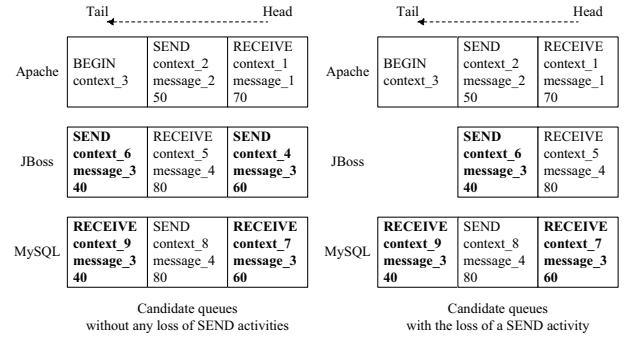


Fig. 6. A case of lost SEND activities.

types of all head activities will be RECEIVE, and our algorithm mentioned above cannot proceed. We take a simple solution to resolve this issue, and we just discard the RECEIVE activity with the smallest timestamp. In Section 5.1.2, our experiments show that our solutions of handling lost activities perform well.

### 3.4.3 The complexity of the algorithm

For a multi-tier service, the time complexity of our algorithm is approximately  $O(g * p * \Delta n)$ , where  $g$  measures the structure complexity of a service,  $p$  is the number of requests in a certain duration, and  $\Delta n$  is the size of activity sequence per request in a sliding time window. Furthermore, the time complexity of our algorithm can be expressed as  $O(g * n)$ , where  $n$  is the size of activity sequence in a sliding time window. The space complexity of our algorithm is approximately  $O(2g * p * \Delta n)$  or  $O(2g * n)$ .

## 4 PRECISETRACER IMPLEMENTATION

We have implemented PreciseTracer with three key components: *TCP\_Tracer*, *Correlator*, *Analyzer*.

After the kernel module named LOG\_TRACE (which is a part of Gather) is loaded, a logging point will be trapped to generate an activity log whenever an application sends or receives a message. The original format of an activity log produced by LOG\_TRACE is "*timestamp hostname program\_name Process ID Thread ID SEND/RECEIVE sender\_ip: port-receiver\_ip: port message\_size*". Gather further transforms the original logs into more informative n-ary tuples to describe the context and message identifier of each activity (described in Section 3.1). Determining activity types is straightforward: SEND and RECEIVE activities are transformed directly; BEGIN or END activities are determined by the port of the communication channel. For example, the RECEIVE activity from a client to the web server's port 80 means the START of a request, and the SEND activity via the same connection in opposite direction means END of a request. After all Gathers have finished log transformation, Coordinator will start *Correlator* and *Analyzer* to further process the collected logs.

Correlator constructs CAGs and delivers them to Analyzer. Analyzer analyzes CAGs to obtain causal path patterns, and further derives statistical information about those patterns.

## 5 EVALUATION

In this section, we evaluate and compare PreciseTracer with respect to WAP5. First, we evaluate and compare PreciseTracer’s accuracy, and efficiency with respect to WAP5. Second, we present PreciseTracer’s sampling effect. Finally, we demonstrate how PreciseTracer can be used to aide in performance debugging. Please note that we perform experiments on three different hardware configurations, indicating that PreciseTracer is independent on specific systems though it needs to instrument the operating system kernel.

### 5.1 Evaluation and comparison of PreciseTracer and WAP5

#### 5.1.1 Experimental setup

We have performed experiments with RUBiS and TPC-W [34]. In the rest of this section, we present detailed results from the RUBiS experiments. Developed at Rice University, RUBiS is a three-tier auction site prototype modeled after eBay.com for evaluating the performance of multi-tier applications.

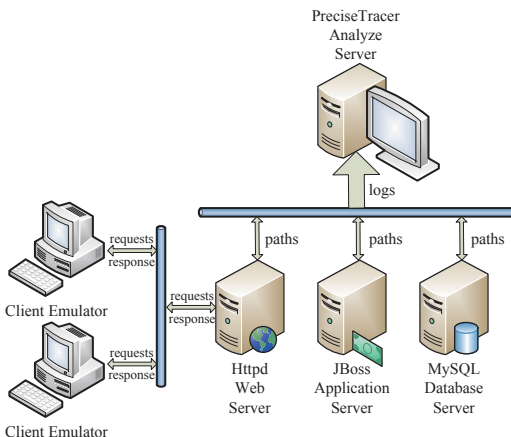


Fig. 7. The deployment diagram of RUBiS.

The experiment platform is a 6-node Linux cluster connected by a 100Mbps Ethernet switch. Apache, JBoss, MySQL and two client emulators are deployed on five SMP nodes with four 1.60GHz Intel(R) Xeon(R) processors and 4GB memory, respectively. The analysis components of PreciseTracer are deployed on a SMP node with two AMD Opteron(tm) processors and 2GB memory. Each node runs the Redhat Fedora Core 6 Linux with the kprobe [31] feature enabled. The deployment of RUBiS is shown in Fig. 7.

Before each experiment, we use NTP to synchronize clocks of three nodes deployed with RUBiS applications, since a NTP service can guarantee time synchronization

to a large extent. For example, NTPv4 can achieve an accuracy of 200 microseconds or better in local area networks under ideal conditions.

In the following experiments, the clients emulate two types of workload: the read\_only workload (Browse\_only) and the read\_write mixed workload (Default). We utilize two physical nodes to emulate the clients. On each node, we set Client Emulator with the same number of concurrent clients. According to the user guide of RUBiS, each workload includes three stages: up ramp, runtime session, and down ramp. In the following experiences, we set different durations for the three stages.

#### 5.1.2 Evaluating accuracies

To compare the accuracies of PreciseTracer and WAP5, we build another library-interposition tool to log network communications. We instrument the following system library functions: *write*, *writetv*, *send*, *read*, *recv*. When a message is sent, a global request ID will be tagged to and propagated with it, and hence we can obtain causal paths with 100% accuracy. The following attributes are logged for the Apache web server, the JBoss Server and the MySQL database: (1) request ID, (2) start time and end time of serving a request; (3) process or thread ID. At the same time, without application-specific knowledge, we use PreciseTracer or WAP5 to identify causal paths and obtain derived information items (2) and (3). If all attributes of a causal path obtained by PreciseTracer or WAP5 are consistent with those obtained by the library-interposition tool, we will confirm that the causal path is correct. Hence we define the path accuracy as:

$$\text{Path accuracy} = \text{correct paths} / \text{all logged requests}$$

We test the accuracy of our algorithm in the offline mode for the read\_write mixed workload of RUBiS. The sliding time window is 400 milliseconds. The number of concurrent clients is set to 100, 300 and 500, respectively. For RUBiS, the up ramp, runtime session, and down ramp durations are set to 1 minute, 0.5 minutes and 1 minute, respectively. Table 1 summarizes the results. Our PreciseTracer outperforms WAP5. PreciseTracer accurately correlates all *logged* activities into causal paths with the accuracy of our algorithm reaching close to 100%, while the accuracy of WAP5 is lower than 70%. The reason for the (minor) imperfection lies in the implementation of SystemTap, which fails to collect the small fraction less than 1% of send/receive activities. For PreciseTracer, Fig. 8 in the subsequent experiments with the number of concurrent clients varying from 100 to 500 shows the sliding time has little effect on the the accuracy of PreciseTracer.

Since we adopt a sampling policy to reduce the system overhead, we cannot accurately keep complete logs for each request of interest. Furthermore, we perform online request tracing to compare accuracies of PreciseTracer and WAP5 v.s. different sampling rates, which can be defined as the ratio of the collecting time window to the sleeping time window. we use a tuple (*online*, *the*

TABLE 1  
CAGs' Accuracy Results

	100	300	500
Total CAGs (library interposition)	5443	16115	26723
Matched (PreciseTracer)	5442	16081	26702
Matched (WAP5)	3710	9634	15707
Accuracy (PreciseTracer)	99.98%	99.79%	99.92%
Accuracy (WAP5)	68.16%	59.78%	58.78%

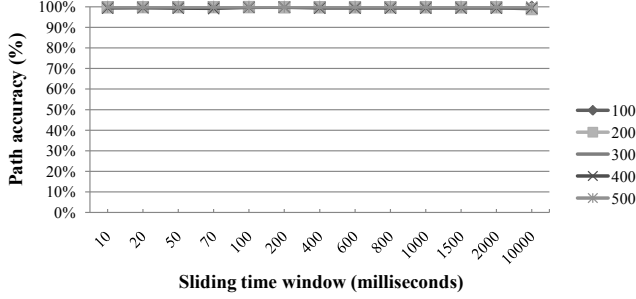


Fig. 8. The path accuracy of PreciseTracer v.s. the sliding time window. Five lines overlaps with each other.

collecting time window, the sleeping time, the sliding time window, rounds) to denote the configuration of online request tracing. Note that in the rest of this section, the unit of sliding time window is millisecond, and the unit of other time metrics is seconds. We choose the fixed fixing concurrent clients as 200. For the configuration (online,  $x$ ,  $y$ , 400, 1) with the varying collecting time window  $x$  and sleeping time  $y$ , we evaluate the accuracies of PreciseTracer and WAP5. Fig. 9 shows that even when the sampling rate is lower as 4%, PreciseTracer still achieves the high accuracy close to 100%, while the accuracy of WAP5 is lower than 60%.

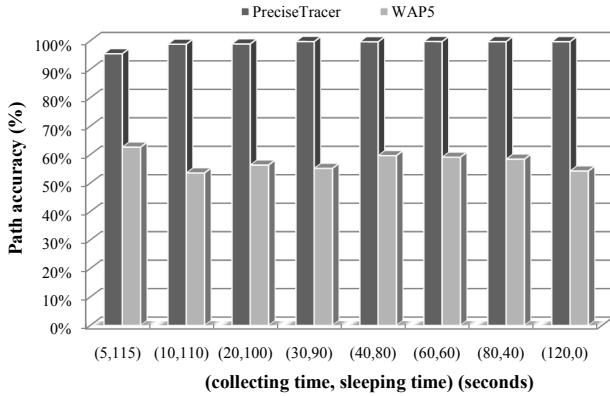


Fig. 9. The path accuracies of PreciseTracer and WAP5 v.s. different sampling rates for online request tracing. The up ramp, runtime session, and down ramp are 1 minute, 0.5 minute and 1 minute, respectively.

### 5.1.3 Evaluating the complexity

In this experiment, the configuration is (online, 150, 600, 400, 1), indicating that PreciseTracer will work under an online mode; it will gather logs for one round which lasts 150 seconds, and then stop collecting logs in the next 600 seconds; the sliding time window is 400 milliseconds.

With the number of concurrent clients varying from 50 to 500, we record the number of requests and the correlation time. The test duration is fixed for the read\_write mixed workload. We set the up ramp, runtime session, and down ramp durations to 1 minute, 10 minute, and 1 minute, respectively.

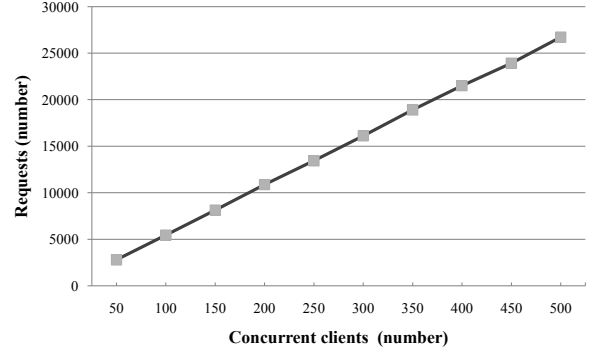


Fig. 10. The request number v.s. the number of concurrent clients.

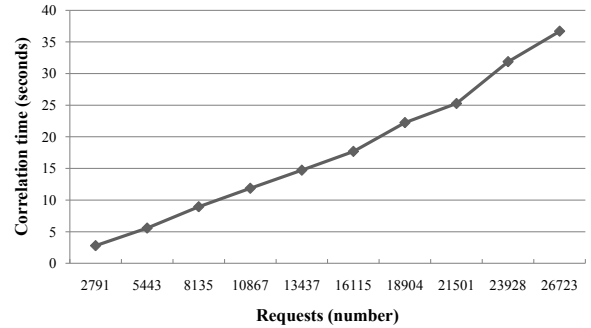


Fig. 11. The correlation time of PreciseTracer v.s. the number of requests.

From Fig. 10 and 11, we observe that the number of requests is almost linear with the number of concurrent clients and the correlation time is almost linear with the number of requests in the fixed duration. In Section 3.4.3, we conclude that the time complexity of our algorithm is  $O(g * p * \Delta n)$ . Our results in Fig. 11 are consistent with that analysis. Since  $g$  is a constant for RUBiS and  $\Delta n$  is unchanged in the fixed sliding time window, the correlation time is linear with the number of requests in the fixed test.

We also compare PreciseTracer with WAP5 on the regard of the correlation time. In Fig.12, for the worst case, the correlation time of WAP5 is 26 times that of PreciseTracer. Fig. 13 shows a partially enlarged view.

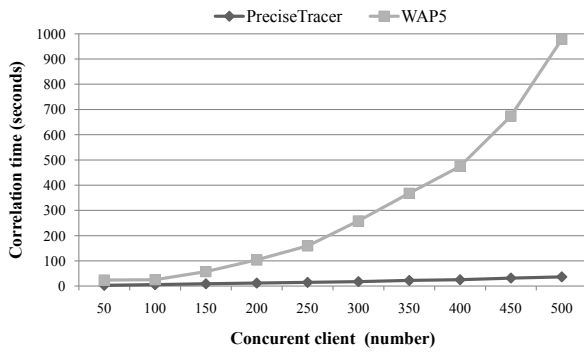


Fig. 12. The correlation time of PreciseTracer v.s. the number of clients.

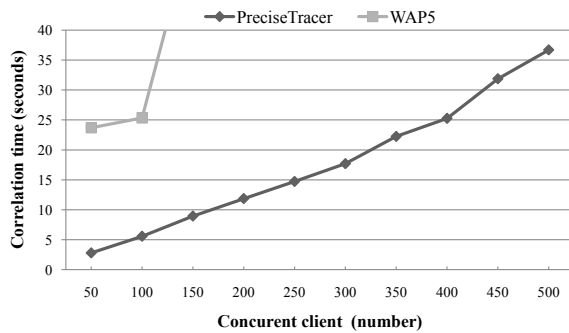


Fig. 13. The partially enlarged view of the correlation time of PreciseTracer v.s. the number of clients.



Fig. 14. The correlation time v.s. the size of the sliding time window. The concurrent clients varies from 50 to 450.



Fig. 15. The partially enlarged view of the correlation time v.s. the size of the sliding time window.

Two important parameters in the experiments could influence the efficiency of PreciseTracer's correlation algorithm: *collecting time window* and *sliding time window*. The collecting time window is the duration of log collection. The longer the collecting time window, the more logs our algorithm has to process thus the higher overhead. The influence of the sliding time window is more complex. Fig. 14 shows the effect of sliding time window size on the correlation time for different number of concurrent clients. Fig. 15 shows a partially enlarged view. The size of sliding time window affects the correlation time in two ways: First, when the number of requests in the fixed duration is fixed, the time complexity of the algorithm is linear with the size of  $\Delta n$  for RUBiS.  $\Delta n$  is the size of log sequence per request in the sliding time window. Hence the correlation time will increase with the sliding time window; Second, if the size of the sliding time window is smaller (less than 10 milliseconds), the following situation will arise more frequently: the related logs of each component fail to be fetched to the buffer of Ranker at the same time hence increasing the correlation time. From Fig. 15, we can observe these effects.

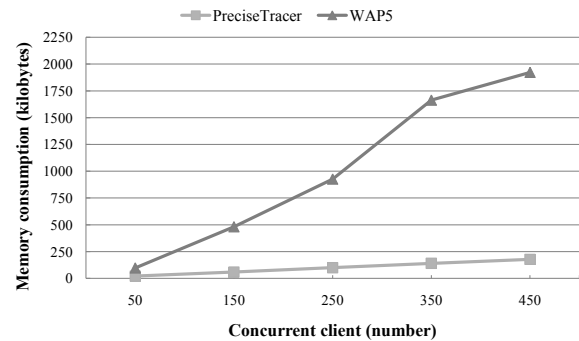


Fig. 16. The memory consumptions of PreciseTracer (the sliding time window is 400 milliseconds) and WAP5.

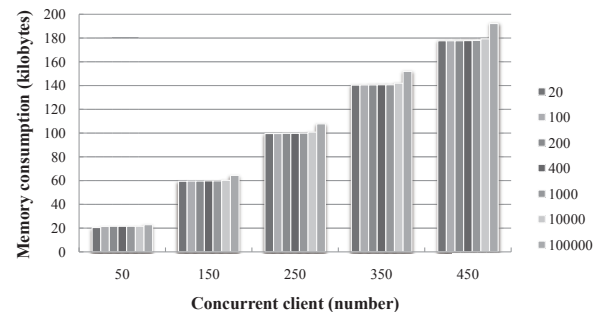


Fig. 17. The memory consumption of PreciseTracer v.s. the size of sliding time window varying from 20 to 100000 milliseconds.

Fig. 16 compares the memory consumptions of PreciseTracer with that of WAP5, indicating that the major analysis component of WAP5 consumes more memory than that of PreciseTracer. Fig. 17 presents the effect of

the size of the sliding time window on the memory consumption of the major analysis module—*Correlator* for different concurrent clients. We do not show the memory consumption of other components of PreciseTracer, because they consume fewer memory in comparison with *Correlator*.

#### 5.1.4 The overhead on the application

We compare the throughput and the average response time of RUBiS for the read\_write mixed workload when the instrumentation mechanism is disabled (no instrumentation), enabled (only starting the Gather module), or PreciseTracer is in the *online analysis* mode. In order to test the worst-case overhead of PreciseTracer under the online model, we choose the continuous collection mode. The configuration of PreciseTracer is (online, 60, 0, 400, 10). We set the up ramp, the runtime session, and the down ramp durations to 1 minutes, 10 minutes, and 1 minute, respectively.

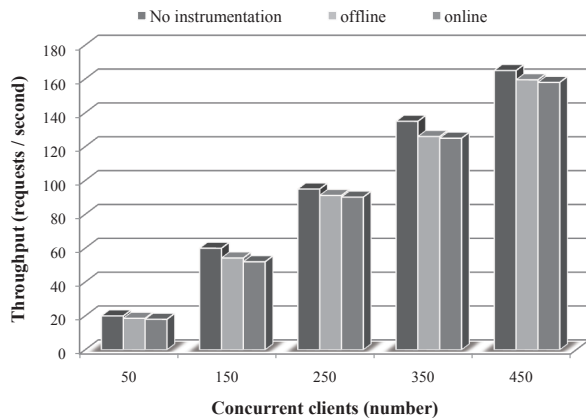


Fig. 18. The effects of PreciseTracer on the throughput of RUBiS.

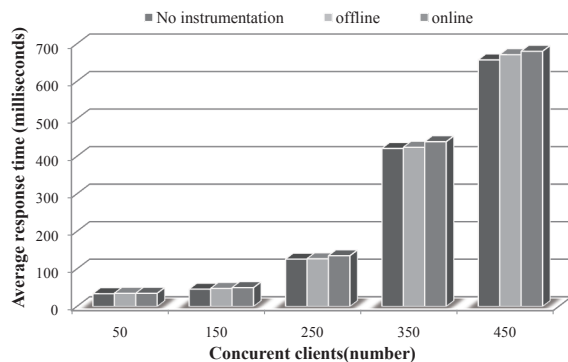


Fig. 19. The effect of PreciseTracer on the average response time.

In Fig. 18 and Fig. 19, we observe that PreciseTracer in the continuous log collection mode has little impact on the throughput and a small and acceptable impact on the average response time of RUBiS.

Note that the overheads of WAP5 on the throughput and the average response time of the application are same as that of PreciseTracer in the offline mode shown in Fig. 18 and Fig. 19 as they both use the same library interposition for collecting logs in our experiments.

#### 5.1.5 Evaluating the online analysis ability

In this section, we demonstrate the online analysis ability of PreciseTracer. We set the baseline configuration as (online, 10, 50, 400, 10). The test duration is fixed for the read\_write mixed workload. We set the up ramp, runtime session, and the down ramp durations to 1 minutes, 5 minutes, and 1 minute, respectively.

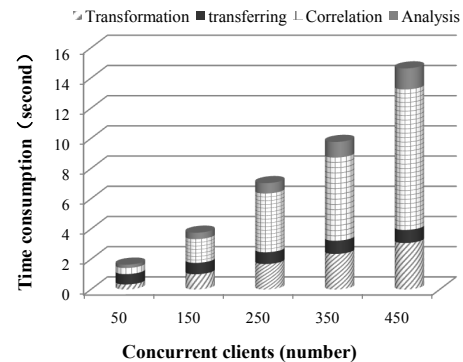


Fig. 20. The time consumptions in each stage of PreciseTracer.

There are four steps in applying request tracing to online analysis of services: first, the system transforms original logs into tuple logs (*transformation*); second, it transfers the tuple logs to an *analysis node* for further analysis (*transferring*); third, the system correlates the tuple logs to compute CAGs (*correlation*); fourth, the system analyzes the CAGs (*analysis*). Because the second step is conducted on each node simultaneously, we take the maximum one as the time consumption in this stage.

Fig. 20 shows that PreciseTracer processes the logs efficiently. In particular, it generates analysis results within 20 seconds when we set concurrent clients to 450. The results demonstrate the practicality of using PreciseTracer for online analysis.

## 5.2 The sampling effect

The experiment platform is a 6-node Linux cluster connected by a 100Mbps Ethernet switch. Apache and JBoss are deployed on two SMP nodes with two PIII processors and 2GB memory, respectively. Database (MySQL) and the analysis components of PreciseTracer are deployed on two SMP nodes with eight Intel Xeon processors and 8GB memory, respectively. Each node runs the Redhat Fedora Core 6 Linux with the kprobe [31] feature enabled.

PreciseTracer supports sampling as it tolerates loss of activity logs. In this section, we demonstrate that adopting a lossy sampling policy could decrease the

size of collected logs, while still capturing most of the dominating causal path patterns.

We run *offline* experiments twice – in one minute and in 10 minutes. The test duration is fixed for the *brown\_only* workload. We set the up ramp, the runtime session and the down ramp durations to 30 seconds, 30 minutes, and 1 minute, respectively. The sliding time window is 20 milliseconds.

We analyze the top 10 dominated causal path patterns in two runs of experiments: 1 minute vs. 10 minutes. We have two observations: first, in two runs, causal paths belonging to the top 10 causal patterns take up a significant percentage of all causal paths (both 88%); Second, two runs of the experiment lead to the *same* top 10 dominated causal path patterns. These two observations justify the sampling policy since adopting a sampling policy still captures most of the dominated casual path patterns.

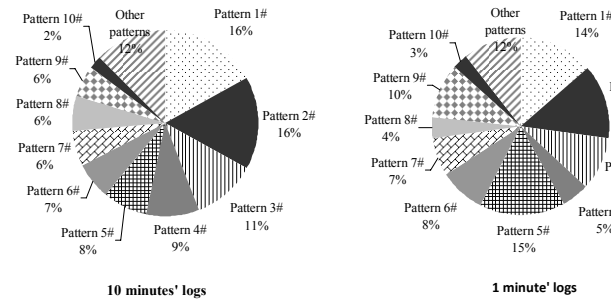


Fig. 21. Comparisons of the dominated causal path patterns in two runs of one minute and ten minutes respectively.

Table 2 compares log sizes in two runs of experiments of 1 minute and 10 minutes, respectively. It indicates that the sampling policy decreases the cost of collecting and analyzing logs, and hence improves system scalability.

TABLE 2  
The log size in two runs of experiments

	Original logs of Apache (M)	Original logs of Jboss (M)	Original logs of MySql (M)
1-minute run	3.9	6.6	6.7
10-minute run	27.9	55.1	58.6

### 5.3 Identifying performance bottleneck

In the following section, we will utilize PreciseTracer to detect performance problems of a multi-tier service. The experiment platform is a Linux cluster connected by a 100Mbps Ethernet switch. Web tier (Apache) and active web pages server (JBoss), database (MySQL) and analysis components of PreciseTracer are deployed on four SMP nodes, respectively, each of which has two PIII processors and 2GB memory. Each node runs the Redhat Fedora Core 6 Linux with the kprobe [31] feature enabled. The deployment of RUBiS is similar to Fig. 7.

#### 5.3.1 Misconfiguration inference

When we perform experiments to evaluate the overhead of PreciseTracer under the offline model, we observe that when the number of concurrent clients increases from 700 to 800, the throughput of RUBiS decreases. Fig. 22 shows the relationship between the number of requests served and the number of concurrent clients. The test duration is fixed for the *brown\_only* workload. And we set the up ramp, the runtime session and the down ramp durations to 1 minute, 7 minutes, and 1 minute, respectively. The sliding time window is 20 milliseconds. An interesting question we try to answer is “what is the root cause for the throughput decline?”

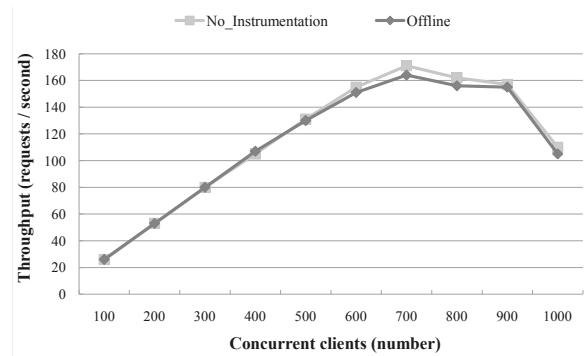


Fig. 22. The overhead of PreciseTracer in the offline model

Generally, we will observe the resource utilization rates of each tier and the metrics of quality of service to pinpoint bottlenecks. Using the monitoring tool of RUBiS, we observe that the CPU utilization rate of each node is less than 80% and the I/O usage rate is not high. Obviously, the traditional method does not help.

To answer this question, we use our tool to analyze the most frequent request —ViewItem for RUBiS, and visualize the view of *latency percentages of each tier and each interaction between tiers*.

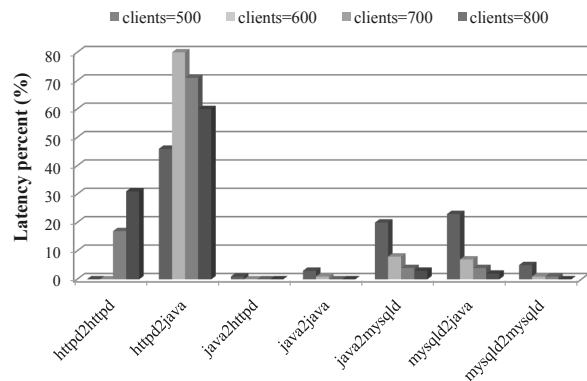


Fig. 23. The latency percentages of each tier and each interaction between tiers.

From Fig. 23, we observe that when the number of concurrent clients increases from 500 to more, the

latency percentage of httpd2Java from the first tier to the second tier changes dramatically, and they are 46%, 80%, 71% and 60%, respectively, for 500, 600, 700 and 800 concurrent clients. In Fig. 23, the latency percentage of httpd2Java is 46% for 500 clients, which means that the processing time for the interaction from httpd to Java takes up the maximum percentage of the end-to-end time of servicing a request.

At the same time, the latency percentage of httpd2httpd (the first tier) increases dramatically from 17% (700 clients) to 31% (800 clients). We observe the CPU utilization rate of the Jboss node is less than 60% and the I/O usage rate is not high. When servicing a request, httpd2httpd is before httpd2java in a causal path. So we can confirm that there is something wrong with the interaction between httpd and JBoss. Through reading the manual of RUBiS, we infer that the problem is mostly likely related to the thread pool configuration of JBoss. According to the manual of JBoss, one parameter named *MaxThreads* controls the maximum available threads, of which each thread serves a connection. The default value of *MaxThreads* is 40.

We set the value of *MaxThreads* as 250 and run the experiments again. In Fig. 24, we observe that our troubleshooting is effective. In Fig. 24,  $TP\_MT_x$  is the throughput when *MaxThreads* is  $x$ , and  $RT\_MT_x$  is the average response time when *MaxThreads* is  $x$ . With the concurrent clients increasing from 500 to 800, the throughput gets higher than under the default configuration (i.e. *MaxThreads* being 40); and the average response time is lower than under the default configuration. However, for 900 concurrent clients, the resource limit of hardware platform results in a new bottleneck, which narrows the performance difference.

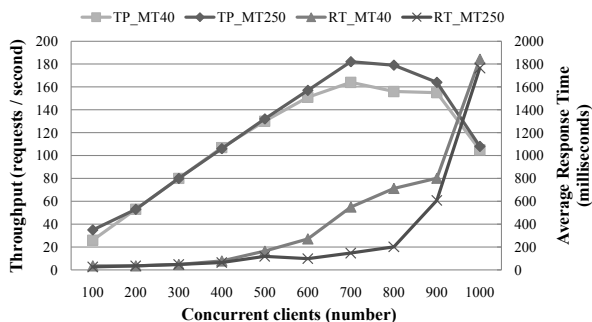


Fig. 24. Performance under different *MaxThreads* configuration

### 5.3.2 Identifying injected performance problems

To further validate the accuracy of locating performance problems using PreciseTracer, we have injected several performance problems into RUBiS components and their host nodes: for abnormal case 1, we modify the code of the second tier to inject a random delay; for abnormal case 2, we lock the items table of the database to inject a delay; for abnormal case 3, we change the configuration

of the Ethernet driver on the node running JBoss from 100Mbps to 10Mbps.

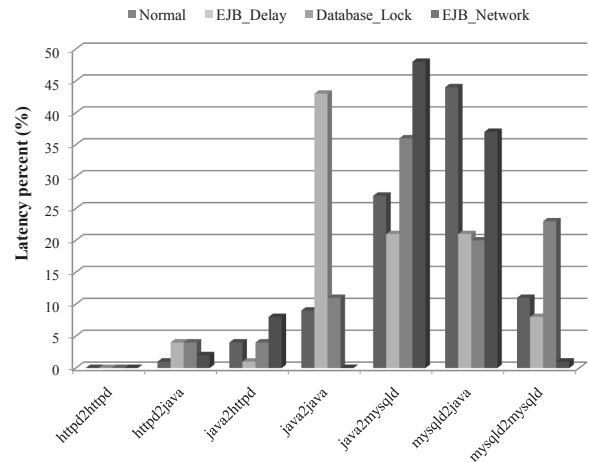


Fig. 25. Latency percentages of components for abnormal cases. Work mode: offline; collecting time window: 10 minutes, sliding time window: 20ms

We use PreciseTracer to locate the component in trouble where different performance problems are injected. Fig. 25 shows the latency percentages of components for normal case and three abnormal cases.

For abnormal case 1 (EJB\_Delay), the latency percentage of Java2Java (the second tier) increases from less than 10% under the normal configuration to more than 40%, and the latency percentages of other components decrease. Hence we infer that JBoss is the problematic component.

For abnormal case 2 (DataBase\_Lock), the latency percentage of mysqld2mysqld (the third tier) increases from 12% under the normal configuration to more than 20%, and the latency percentage of java2mysqld (interaction from the second tier to the third tier) increases from 26% to more than 35%. The Latency percentages of other components remain unchanged or decrease. Hence we infer that MySQL is in trouble.

For the abnormal case 3 (EJB\_Network), the latency percentage of Java2mysqld (from the second tier to the third tier) increases from 26% to 47%; mysqld2java (from the third tier to the second tier) remains at about 37%. The latency percentage of httpd2java from the first tier to the second tier increases from 1% to 2%; the percentage of java2httpd from the second tier to the first tier increases from 4% to 8%. We observe that most of time for servicing a request is spent on the interactions between the second tier and the third tier, and the three latency percentages out of the four interactions with the second tier increase. We infer that the second tier has problem. Further observation shows the latency percentage of Java2java strangely decreases from 9% to almost 0%. So we further conclude that there is something wrong with the network of the second tier.

## 6 RELATED WORK

This section summarizes related work from three perspectives: black-box and white-box tracing approaches, and profiling approaches.

### 6.1 Black-box tracing approaches

TABLE 3  
Comparisons of black-box tracing approaches

	<i>accuracy</i>	<i>scalability</i>	<i>request abstraction</i>	<i>mode</i>
<i>Project5</i>	imprecise	/	macro level	offline
<i>WAP5</i>	imprecise	/	macro level	offline
<i>E2EProf</i>	imprecise	/	macro level	online
<i>BorderPatrol</i>	precise	continuous logging	micro level	offline
<i>vPath</i>	precise	continuous logging	micro level	offline
<i>PreciseTracer</i>	precise	sampling & tracing on demand	micro/macro levels	online

#### 6.1.1 Imprecise black-box tracing approaches

A much earlier project, *DPM* [9], instruments the operating system kernel and tracks the causality between pairs of messages to trace unmodified applications. However, *DPM* is not precise, since any real causal path does not necessarily follow the edges of a path in *DPM*'s output graph. *Project5* [4] and *WAP5* [3] accept imprecision of probabilistic correlations. *Project5* proposes two algorithms for offline analysis. A nesting algorithm assumes 'RPC-style' (call-returns) communication, and a convolution algorithm does not assure a particular messaging protocol. The nesting algorithm only uses one timestamp per message [3] without distinguishing SEND or RECEIVE timestamp, and hence it only provides aggregate information per component. The convolution algorithm only infers average response time of components, and cannot build individual causal paths (micro-level request abstraction) for each request. More recently, *WAP5* [3] infers causal paths from tracing stream on a per-process granularity via library interposition, and propose a message-linking algorithm for inferring causal relationships between messages. Similar to the convolution algorithm of *Project5*, *E2EProf* [7] proposes a pathmap algorithm, and uses compact trace representations and a series of optimizations to make itself suitable for online performance diagnosis. Just like *Project5* and *WAP5*, *E2EProf* is inaccurate.

#### 6.1.2 Precise black-box tracing approaches

There are only two precise black-box approaches, *BorderPatrol* and *vPath*, which are offline and cannot offer performance information in real time. With the knowledge of diverse protocols used by multi-tier service, *BorderPatrol* isolates and schedules events or requests at the

protocol level to precisely trace requests. When multi-tier services are developed from commercial components or heterogeneous middleware, *BorderPatrol* has to write many protocol processors and requires more specialized knowledge than pure black-box approach [5]. *vPath* consists of a monitor and an offline log analyzer. The monitor continuously records which thread performs a send or recv system call over which TCP connection. The offline log analyzer parses logs generated by the monitor to discover request processing paths. The monitor is implemented in virtual machine monitor (VMM) through system call interceptions. Using library interposition (*BorderPatrol*) or system call interception (*vPath*), the logging mechanism of *BorderPatrol* or *vPath* cannot be enabled or disabled on demand without interrupting services, and hence it is unscalable in terms of the cost of collecting and analyzing logs. For example, as stated in [2], a simple e-commercial system could generate 10M logs per minute. A data center usually consists of tens of thousands or even more nodes being deployed with multi-tier services. If we debug performance problems of multi-tier services on this scale for one minute, a tracing system with continuous logging has to analyze at least 0.1TB logs, which is unscalable. Besides, *BorderPatrol* and *vPath* fail to present macro-level abstractions to facilitate debugging performance-in-the-large, and users have to deal with massive logs with great efforts. With respect to our previous work [20], our new contributions are two-fold. First, we improved the tracing algorithm to tolerate log losses, on the basis of which, we developed two mechanisms: tracing on demand and sampling to significantly increase system scalability. Through experiments, we demonstrate that adopting a sampling policy could decrease the size of collected logs, while still preserving performance data of services in the way that it captures most of dominated causal path patterns. Second, we designed and implemented an online request tracing tool. Through experiments, we demonstrate the online analysis ability of *PreciseTracer*.

### 6.2 White-box tracing approaches

The common requirements of white-box approaches are that they need to obtain the source code of applications or middleware.

The most invasive systems, such as *Netlogger* [10] and *ETE* [11], require programmers to add event logging to carefully-chosen points to locate causal paths, rather than infer them from passive traces. *Pip* [13] inserts annotations into source code to record actual system behaviors, and can extract causal path information with no false positives or false negatives. *Magpie* [1] collects events at different points in a system and uses an event schema to correlate these events into causal paths. In order to track a request from end to end, *Magpie* must obtain the source code of an application, at least require "wrapper" around some parts of the application [4]. *Stardust* [12] is a system used as an on-line monitoring

tool in a distributed storage system, and is implemented in a similar manner. *Whodunit* [14] annotates profile data with transaction context synopsis, tracks and profiles transactions that flow through shared memory, events, and inter-process communication. Chen *et al.* [19] applies path-based macro analysis to two broad classes of tasks encountered with large distributed systems: failure management and evolution. Their approach is to associate each request with a unique identifier at the system entry point while maintaining the association throughout the process.

To avoid modifying applications' source code, several previous work has enforced middleware or infrastructure changes, bound to specific middleware or deployed instrumented infrastructure. *Pinpoint* [15] locates component faults in J2EE platforms by tagging and propagating a globally unique request ID with each request. *Causeway* [16] enforces change to network protocol so as to tag meta-data with existing module communication. *X-Trace* [17] modifies network layer to carry X-Trace meta-data, which enables casual path reconstruction and focuses on debugging paths through network layer. As Google's production tracing infrastructure, *Dapper* [27] uses a global identifier to tie related events together from various parts of a distributed system, which mandates accessing the source code of applications. Dapper uses sampling to improve system scalability and reduce performance overhead.

With the support of logging mechanism of Hadoop, Tan *et al.* [28] presented a non-intrusive approach to tracing the causality of execution in MapReduce-like cloud systems [36], which is significantly different from multi-tier services.

### 6.3 Profiling approaches

Tracing is a measurement of a stream of events of the behavior of a system [39], while profiling is measurement of a statistical summary of the behavior of a system. Another tool from Google, Google-Wide-Profiling(GWP) [35], a continuous profiling infrastructure for data centers, provides performance insights for cloud applications. GWP introduces novel applications of its profiles, such as application platform affinity measurements and identification of platform-specific, micro-architectural peculiarities. In our previous work [40], we design and implement an innovative system, AutoAnalyzer, that automates the process of debug performance problems of SPMD-style parallel programs, including data collection, performance behavior analysis, locating bottlenecks, and uncovering their root causes. AutoAnalyzer is unique in terms of two features: first, without any apriori knowledge, it automatically locates bottlenecks and uncovers their root causes for performance optimization; second, it is lightweight in terms of the size of performance data to be collected and analyzed.

## 7 CONCLUSION

We have developed an accurate request tracing tool, called *PreciseTracer*, to help users understand and debug performance problems in a multi-tier service of black boxes. Our contributions lie in four-fold: (1) we have designed a precise tracing algorithm to derive causal paths for each individual request, which only uses application-independent knowledge, such as timestamps and end-to-end communication channels; (2) we have presented two abstractions, component activity graph and dominated causal path pattern, for understanding and debugging micro-level and macro-level user request behaviors of the services, respectively; (3) we have developed two mechanisms, tracing on demand and sampling, to increase the system scalability; and (4) we have designed and implemented an online request tracing system. To validate the efficacy of *PreciseTracer*, we have conducted extensive experiments on 3-tier platforms. In comparison with WAP5—a black-box tracing approach, *PreciseTracer* achieves higher tracing accuracy and faster response time. Our experimental results also show *PreciseTracer* has low overhead, and still achieves high tracing accuracy even if an aggressive sampling policy is adopted, which indicates that *PreciseTracer* is a promising tracing tool for large-scale production systems.

## ACKNOWLEDGMENT

We are very grateful to anonymous reviewers. This work is supported by the Chinese 973 project (Grant No.2011CB302500) and the NSFC project (Grant No.60933003).

## REFERENCES

- [1] P. Barham, *et al.* *Using Magpie for Request Extraction and Workload Modeling*. In Proc. 6th OSDI, 2004, pp. 18-18.
- [2] P. Barham, *et al.* *Magpie: online modelling and performance-aware system*. In Workshop on HotOS'03, 2003, PP.85-90.
- [3] P. Reynolds, *et al.* *WAP5: Black-box Performance Debugging for Wide-area Systems*. In Proc. 15th WWW, 2006, pp.347-356.
- [4] M. K. Aguilera, *et al.* *Performance Debugging for Distributed Systems of Black Boxes*. In Proc. 19th SOSP, 2003, pp. 74-89.
- [5] E. Koskinenand, *et al.* *BorderPatrol: Isolating Events for Black-box Tracing*. SIGOPS Oper. Syst. Rev. 42, 4, 2008, pp. 191-203.
- [6] M. Ricahrd Stevens, *UNIX Network Programming Networking APIs: Sockets and XTI, Volume 1*, Prentice Hall, 1998.
- [7] S. Agarwala, *et al.* *E2EProf: Automated End-to-End Performance Management for Enterprise Systems*. In Proc. 37th DSN, 2007, pp.749-758.
- [8] L. Lamport. *Time, Clocks and the Ordering of Events in a Distributed System*, Comms. ACM, 21(7), 1978, pp.558-565.
- [9] B. P. Miller. *DPM: A Measurement System for Distributed Programs*, IEEE Trans. on Computers, 37(2), 1988, pp.243-248.
- [10] B. Tierney, *et al.* *The NetLogger Methodology for High Performance Distributed Systems Performance Analysis*. In Proc. 17th HPDC, 1998, pp. 260-267
- [11] J. L. Hellerstein, *et al.* *ETE: a Customizable Approach to Measuring End-to-end Response Times and Their Components in Distributed Systems*. In Proc. 19th ICDCS, 1999, pp. 152-162.
- [12] E. Thereska, *et al.* *Stardust: Tracking Activity in a Distributed Storage System*. In Proc. SIGMETRICS, 2006, pp. 3-14.
- [13] P. Reynolds, *et al.* *Pip: Detecting the Unexpected in Distributed Systems*. In Proc. 3rd NSDI, 2006, pp.115-128.
- [14] A. Chanda,*et al.* *Whodunit: Transactional Profiling for Multi-tier Applications*, SIGOPS Oper. Syst. Rev. 41, 3, 2007, pp. 17-30.

- [15] M. Y. Chen, et al. *Pinpoint: Problem Determination in Large, Dynamic Internet Services*. In Proc. 32th DSN, 2002, pp.595-604.
- [16] A. Chanda, et al. *Causeway: Operating System Support for Controlling and Analyzing the Execution of Distributed Programs*. In Proc. 10th HotOS, 2005, pp. 18-18.
- [17] R. Fonseca, et al. *X-Trace: A Pervasive Network Tracing Framework*. In Proc. 4th NSDI, 2007, pp.271-284.
- [18] A. Anandkumar, et al. *Tracking in a spaghetti bowl: monitoring transactions using footprints*. In Proc. SIGMETRICS'08, 2008, pp. 133-144.
- [19] M. Y. Chen, et al. *Path-based failure and evolution management*. In Proc. NSDI'04, 2004.
- [20] Z. Zhang, et al. *Precise Request Tracing and Performance Debugging of Multi-tier Services of Black Boxes*. In Proc. DSN'09, 2009, pp.337-346.
- [21] B. Sang, et al. *Decreasing Log Data of Multi-tier Services for Effective Request Tracing*. In Proc. DSN'09 Fast Abstract.
- [22] B. C. Tak, et al. *vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities*. In Proc. USENIX'09, 2009.
- [23] B. M. Cantrill et al. *Dynamic Instrumentation of Production Systems*. In Proc. USENIX'04, 2004.
- [24] Y. Ruan et al. *Making the "box" transparent: system call performance as a first-class result*. In Proc. USENIX ATC'04, 2004.
- [25] Y. Ruan et al. *Understanding and Addressing Blocking-Induced Network Server Latency*. In Proc. of the USENIX ATC' 06, 2006.
- [26] K. Shen, et al. *Hardware counter driven on-the-fly request signatures*. In Proc. ASPLOS XIII, 2008, pp.189-200.
- [27] B. H. Sigelman, et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*, Google Technical Report dapper-2010-1, April 2010.
- [28] J. Tan, et al. *Visual Log-based Causal Tracing for Performance Debugging of MapReduce Systems*. In Proc. of ICDCS'10.
- [29] C.Stewart et al. *Performance Modeling and System Management for Multi-component Online Services*. In Proc.NSDI'05, 2005.
- [30] K. Appleby, et al. *Oceano- SLA Based Management of a Computing Utility*. In Proc. of the IFIP/IEEE Symposium on Integrated Network Management, 2001, pp.855-868.
- [31] R. Krishnakumar. *Kernel kerner: kprobes-a kernel debugger*. Linux J. 2005, 133 (May. 2005), 11.
- [32] L. Yuan, et al. *PowerTracer, tracing requests in multi-tier services to save cluster power consumption*. Technical Report. 2010, Available at <http://arxiv.org/corr/>.
- [33] SystemTap: <http://sourceware.org/systemtap>
- [34] TPC Benchmark: <http://www.tpc.org/tpcw/>
- [35] G. Ren, et al. *Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers*. IEEE Micro 30, 4 (July 2010), 65-79.
- [36] P. Wang, et al. *Transformer: A New Paradigm for Building Data-Parallel Programming Models*. IEEE Micro 30, 4 (July 2010), 55-64.
- [37] L. Wang, et al. *In cloud, do MTC or HTC service providers benefit from the economies of scale?*. In Proc. of MTAGS '09. 2009.
- [38] L. Wang, et al. *In cloud, can scientific communities benefit from the economies of scale?*. Accepted by TPDS. March, 2011.
- [39] Readings in Instrumentation, Profiling, and Tracing: <http://www.inf.usi.ch/faculty/hauswirth/teaching/ipt.html>.
- [40] X. Liu, et al. *Automatic Performance Debugging of SPMD-style Parallel Programs*. Accepted by JPDC. March, 2011.

## APPENDIX A PRECISETRACER PSEUDO-CODE

---

### Algorithm 1 Procedure correlate {}

---

```

1. while current=ranker.rank( ) do
2.   e = current → get_type()
3.   if (e == BEGIN) then
4.     create a CAG with current as its root;
5.   else if (e == END) then
6.     find the matched parent where
       parent →c current;
7.     if (the match is found) then
8.       add current into the matched CAG;
9.       add an adjacent context edge from parent to
       current;
10.    output CAG;
11.   end if
12. else if (e == SEND) then
13.   find matched parent_msg where par-
       ent_msg →c current;
14.   if (the match is found) then
15.     if (parent_msg.type == SEND) then
16.       parent_msg.size += current.size;
17.     else
18.       add current into the matched CAG.
19.       add an adjacent context edge from par-
       ent_msg to current.
20.   end if
21. end if
22. else if (e == RECEIVE) then
23.   find matched parent_msg where par-
       ent_msg →m current;
24.   if (the match is found) then
25.     parent_msg.size -= current.size;
26.     if (current → CAG = NULL) then
27.       add current into the matched CAG;
28.       current → CAG = current CAG;
29.       add a message edge from parent_msg to cur-
       rent;
30.     if (matched parent_cntx where
       parent_cntx →c current is found) then
31.       if (parent_msg and parent_cntx are in the
       same CAG) then
32.         add a context edge from parent_cntx to
       current;
33.       end if
34.     end if
35.   end if
36. end if
37. end if
38. end while

```

---



---

### Algorithm 2 Function is\_noise(Activity \*E)

---

```

1: return (E → type == RECEIVE) and (No matched
   SEND activity X in mmap with X →m E) and ( No
   matched SEND activity Y in the buffer of Ranker
   with Y →m E)

```

---