Performance Debugging for Distributed Systems of Black Boxes

Marcos K. Aguilera*
Jeffrey C. Mogul
Janet L. Wiener
HP Labs, Palo Alto
{Marcos.Aguilera,Jeff.Mogul,
Janet.Wiener}@hp.com

Patrick Reynolds
Duke University
reynolds@cs.duke.edu

Athicha Muthitacharoen
MIT Lab for Computer Science
athicha@lcs.mit.edu

ABSTRACT

Many interesting large-scale systems are distributed systems of multiple communicating components. Such systems can be very hard to debug, especially when they exhibit poor performance. The problem becomes much harder when systems are composed of "black-box" components: software from many different (perhaps competing) vendors, usually without source code available. Typical solutions-provider employees are not always skilled or experienced enough to debug these systems efficiently. Our goal is to design tools that enable modestly-skilled programmers (and experts, too) to isolate performance bottlenecks in distributed systems composed of black-box nodes.

We approach this problem by obtaining message-level traces of system activity, as passively as possible and without any knowledge of node internals or message semantics. We have developed two very different algorithms for inferring the dominant causal paths through a distributed system from these traces. One uses timing information from RPC messages to infer inter-call causality; the other uses signal-processing techniques. Our algorithms can ascribe delay to specific nodes on specific causal paths. Unlike previous approaches to similar problems, our approach requires no modifications to applications, middleware, or messages.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*distributed debugging, testing tools*

General Terms

Algorithms, Performance, Measurement

Keywords

Performance debugging, black box systems, distributed systems, performance analysis

*The order of author names is random.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

1. INTRODUCTION

Many commercially-important systems, especially Web-based applications, are composed of a number of communicating components. These are often structured as distributed systems, with components running on different processors or in different processes. For example, a multi-tiered system might start with requests from Web clients that flow through a Web-server front-end and then to a Web "application server," which in turn makes calls to a database server, and perhaps additional services (authentication, name service, credit-card authorization, customer relationship management, etc.).

Such systems can be very hard to debug, especially when they exhibit poor performance. Distributed systems are already hard to debug, but the problem becomes much harder when they are composed of "black-box" components: software from many different (perhaps competing) vendors, usually without source code available.

Enterprises often buy complex systems as complete, customized packages from a "solutions vendor." Solutions vendors must deliver complex component-based systems without the expense of highly-skilled, experienced programmers. While modestly-skilled programmers can design and construct such systems, they may lack the expertise to solve performance problems efficiently. Vendors of individual components provide training and support for solving performance problems within the components, but not necessarily *among* multi-vendor components. Therefore, whole-system performance debugging can require either an inordinate amount of time, or the services of expensive and hard-to-find systems integration experts. Both problems cut into profits for solutions vendors.

We contend that performance-oriented operating systems research must focus on performance-in-the-large, rather than merely delivering incremental improvements for low-level component performance. Complex systems exhibit performance problems that often grow out of the system complexity, and while these can sometimes be solved by improving the performance (or selection) of low-level components, they cannot be *diagnosed* by focusing on the components.

Our goal is to design tools that help programmers isolate performance bottlenecks in black-box distributed systems. These tools should not require much (or any) direct support from the components themselves, because we do not want to assume that software vendors will make any effort to support a particular methodology. The tools will not themselves solve any performance problems, but by isolating problems efficiently and (we hope) accurately, they should increase the efficiency both of modestly-skilled programmers and of experts at systems integration.

In this paper, we describe a specific approach to this goal, based on application-independent passive tracing of communication between the nodes in a distributed system, combined with off-line analysis of these traces. We show that traces gathered with little or no knowledge of application design or message semantics are sufficient to make useful attributions of the sources of system latency. Our insistence on passive tracing with no application modification makes our approach applicable to almost any distributed system, and differentiates our work from other approaches that either require application or middleware modifications, or make stronger assumptions about applications or messages.

2. PROBLEM STATEMENT, GOALS, AND NON-GOALS

We model a distributed system as a graph of communicating nodes. Nodes might be computers, in which case the edges are the network connections between communicating pairs of nodes. (Our approach handles other node granularities, as we will discuss in Section 6.) An external request to the system causes activities in the graph along a *causal path*: a series of node traversals where each traversal is caused by some message from a prior node on the path. (Spontaneous system operations can also generate activities on causal paths.)

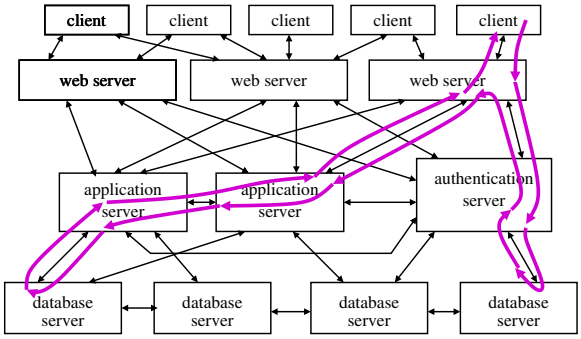


Figure 1: Example multi-tier application showing a causal path

Figure 1 shows an example of a typical distributed system (a multi-tier Web application), with one possible causal path superimposed as a thick line. (The thinner arrows show potential one-hop message paths.) Note that this path flows several times through most of the nodes it touches, since this application is RPC-based and both calls and returns cause node operations.

We assume that all latencies in such a system can be ascribed to the node traversals. (In a system with significant network delays, we can model each long-delay network connection as a pair of zero-delay connections and a virtual delay node. Our initial target environments are LAN-based systems, where network propagation and switching delays are normally negligible.)

Note that the same node may impose different delays for different traversals. For example, the "authentication server" node in Figure 1 is traversed twice in the path shown: once between its invocation and its call to the database, and once between the database's response and its own response to the Web server. Because these are, most probably, different code paths, they could easily have different latencies.

While the example in Figure 1 is an *RPC-style* system, our approach also covers *message-based systems*, in which messages may flow arbitrarily from node to node, without explicit call-return semantics. For example, viewed at the level of an email message, the Internet mail system is message-based, not RPC-style. Simple distance-vector routing protocols are also message-based.

The aim of our project is to create tools and methodologies that enable programmers to understand the sources of latency in a distributed system. In the context of our model, we want these tools to:

- find the high-impact *causal path patterns*—the repeatedly executed causal paths that account for a significant fraction of the system's latency as observed by its users. These are the patterns that are executed frequently and with high mean latency relative to any other frequently-executed patterns.
- identify those nodes on high-impact patterns that, as participants on these patterns, add significant latency to the patterns. For example, an authentication server that caches its results might be used by several nodes in a Web application, but might only cause high latency when invoked from a login-server node, and not when invoked from deeper in the application.

To illustrate the importance of the second point, we use an analogy: a flat procedure profiler can tell you that a function is consuming lots of time, but only a hierarchical profiler (e.g., gprof[9]) can tell you that the problem is not that the function is slow, but that it is being called from a place that doesn't need to call it so often. The context in which a component is used may be of critical importance in diagnosis.

The requirements above define what makes the tools useful. We also impose some requirements to make the tools broadly applicable to systems of black-box components. Our tools should:

- require minimal knowledge (on the part of the tool or the user) of the semantics of the application, the implementation of nodes, the semantics of messages, or *a priori* information about communication paths.
- require no modifications to applications, middleware, messages, or workloads.
- not significantly perturb system performance.

We believe that a tool that requires application-specific knowledge, or application modifications, is much less likely to be used. We especially wish to avoid the need to deploy new infrastructure or promulgate new standards before our tools could be useful. We would like to approximate the ideal of a tool that takes no effort to use. Therefore, one meta-goal for our research is to test how close we can get to the zero-knowledge ideal.

We also have some non-goals:

- We are not developing tools to replace the need for programmers. Performance diagnosis is hard, and our goal is to make it easier for humans, not to automate it.
- Our tools are not meant to verify correct system behavior, or diagnose the causes of faulty behavior.
- Our tools are not aimed at characterizing or benchmarking system performance.
- Because our tools are aimed at the debugging phase, we do not require real-time results; we are willing to use offline analysis (as with a procedure profiler).

2.1 Hypotheses

We are attempting to validate two hypotheses:

1. Our black-box approach is sufficient to identify high-latency causal path patterns with useful precision, and to ascribe the sources of such latency to specific nodes in the context of specific patterns.

2. Given that our black-box approach does identify the sources of latency, this information is useful to a programmer who is debugging the performance of a distributed system.

The first hypothesis can be evaluated using traditional metrics of computer system evaluation. The second hypothesis, however, is an assertion about what humans find useful in carrying out a messy, intellectually challenging task. In this paper, we concentrate on the first hypothesis, and leave the validity of the second to the reader's intuition.

3. RELATED WORK

Before describing our approach, we review several categories of related work.

3.1 Similar approaches to similar problems

Several research projects have attacked the problem of performance debugging in distributed systems, but have taken less radical approaches to the problem of black-box components. In particular, they all require either a homogeneous implementation environment or more intrusive instrumentation. None of these systems rely on passive message tracing.

Hrischuk *et al.* [11] obtain causal traces of distributed computations, including various resource demands (not just latency), by labelling each end-to-end activity using an object-oriented prototyping language (Mlog). Although this did not require modification of the prototype application, their approach is not applicable outside this prototyping system, and in particular would not be useful for systems built from legacy components.

Probably the work closest to ours is Magpie [13], which is also aimed at performance analysis of distributed systems. Magpie, too, treats components as black boxes. However, Magpie specifically associates traced messages with incoming requests, by "tagging incoming requests with a unique identifier and associating resource usage throughout the system with that identifier." This implies a more sophisticated tracing infrastructure than in our approach, but perhaps less need for complex post-processing. Magpie also concentrates more than we do on detecting relatively rare anomalies.

A much earlier project, the Distributed Programs Monitor (DPM) [18], also reports paths of causality through distributed systems. It uses kernel instrumentation to track the causal information between pairs of messages, rather than trying to infer causality from message timestamps. DPM reports an edge between a pair of nodes if any causal path includes that edge. Therefore, the existence of a path in the resulting graph does not necessarily mean that any real causal path followed all of those edges in that sequence.

3.1.1 Commercial products

Several companies already sell software to isolate performance problems using causal tracing. For good commercial reasons, these products aim at a robust solution for a narrow version of the problem we are addressing; our approach is both broader and riskier.

The products that we are aware of concentrate mostly on instrumenting Java applications, since this is a commercially viable market and because the Java Virtual Machine (JVM) provides a convenient locus for non-intrusive instrumentation. (Some systems focus on .Net instead of Java.) They usually also instrument one or more popular non-Java HTTP servers, of necessity, but lack the ability to deal with a broader range of "legacy" nodes.

For example, AppAssure [1] can automatically create component dependency models, using "adapters" that poll components through existing APIs and by instrumenting J2EE method calls. PerformaSure [23] reconstructs execution paths by tagging end-to-end activities (user transactions) as they flow through a J2EE-based system. OptiBench [22] collects traces by instrumenting J2EE and Java interfaces, and apparently can provide fine-grained timing for steps on causal paths. OptiBench also supports transaction replay, providing problem re-creation to aid in debugging.

3.2 Different approaches to similar problems

Tierney *et al.* [26] describe NetLogger, a system for real-time diagnosis of performance problems in distributed systems. Their approach requires programmers to add event logging to carefully-chosen points in the application, and generates "lifelines" that correspond to our causal paths. NetLogger provides tools for managing and visualizing logs, but the tools appear unable to aggregate information from many executions of the same causal path.

Hellerstein *et al.* [10] described ETE, an approach for measuring both end-to-end response times and the contributing component latencies. Their approach requires programmers to instrument applications to reveal significant events and to describe interesting transactions ahead of time, so it is not a black-box technique.

3.3 Similar approaches to different problems

Chen *et al.* [5, 6] describe Pinpoint, a system for locating the components in a distributed system most likely to be the cause of a fault. Pinpoint differs from our work in that they focus on faults rather than performance problems. Their approach involves collecting end-to-end traces of client requests travelling through a distributed system by tagging J2EE calls with a request-ID; this requires no direct application modification, but is currently limited to single-machine tracing. They then use data-mining techniques to correlate low-level faults with high-level problems.

Brown *et al.* [3] also describe a technique aimed at problem (fault) determination based on characterizing dynamic dependencies between components. However, rather than using traces (as in Pinpoint), they perturb system components (for example, by temporarily locking a database table to prevent a component from making progress). Bagchi *et al.* [2] describe a similar approach based on fault injection. Note that the resulting pair-wise dependencies are less specific than end-to-end causal paths would be, and the perturbation approach, which is definitely not passive, requires considerable knowledge of the system design.

In Section 5.2 we describe an algorithm for discovering causality from traces based on statistical correlation. Zhang and Paxson [27] also use statistical techniques, correlating traffic to detect intruders who subvert hosts for use as "stepping stones" (i.e., intruders who telnet into a host and then out of it, intending to cover their tracks). Huang *et al.* use analysis of passively-obtained network traces to detect performance problems in wide-area networks [12]. However, they are interested in network-scale phenomena (delay or congestion) rather than causality.

4. OVERVIEW OF OUR APPROACH

How might a tool that understands nothing about the semantics or implementation of the individual components locate performance problems in a distributed system? Our approach relies on tracing the messages between the nodes, and using one of several offline algorithms to infer causality from these traces. In particular, our algorithms infer multi-hop *causal path patterns*, and provide statistics about latency, both of each pattern and of each node traversal as it occurs in a particular pattern.

The upper part of Figure 2 gives an example of one activation of a simple system. Node A is the "client" that initiates a causal path, which consists of three RPCs (A calls B, B calls C, then calls D, then returns to A). The path includes eleven steps: six RPC messages (odd-numbered arrows) and five intra-node delays (even-

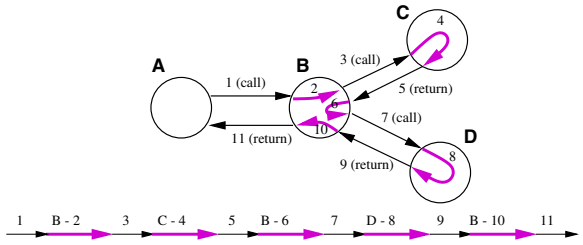


Figure 2: Example causal path in detail

numbered arrows). The lower part of the figure is an abstraction of the causal path showing that the eleven steps are causally related. Since our message traces include only the six odd-numbered steps, our task is to infer the even-numbered steps and the causal connections. We also want to know latencies for the even-numbered steps, which would allow us to report that the latency of the causal path is dominated, for example, by step 6 (internal to node B). *A key feature of our approach is that we can separately report the delays of steps 2, 6, and 10 in this example.*

This task is difficult because a real trace contains interleaved messages from many separate causal paths. Also, a user of our tool wants to see a statistical summary for each of the most important causal path patterns in the system, not a list of every causal path seen in the trace.

Our approach involves three phases:

1. **Exposing and tracing communication:** In this *online* phase, we gather a complete trace of all inter-node messages for an operational system, under real or synthetic load. Depending on the means of communication, we might obtain a single global trace, or a set of per-edge traces for each pair of communicating nodes. This phase creates several logistical problems, including how we obtain individual messages without perturbing the system, how we convert the messages to a concise trace, and how we manage and store large amounts of trace data. Section 6 discusses this phase.
2. **Inferring causal paths and patterns:** In this *offline* phase, we post-process a trace using one of several algorithms. The algorithms must cope with traces that are potentially quite large and noisy (e.g., with missing entries, extraneous calls, timeouts and retries, unsynchronized clocks, etc.). Although this phase need not meet real-time performance goals, our algorithms must be reasonably efficient in time and space. However, the algorithms need not be fool-proof, because our tools are meant to help humans debug systems, not for automatic control. They should be robust enough that they seldom fail with false negatives (i.e., failing to detect the most important causal path patterns) or false positives (overwhelming the user with extraneous information). Section 5 describes our algorithms.
3. **Visualization:** A full system should also provide appropriate ways to visualize the results. However, our research so far has only partially addressed this issue.

An abstract trace format forms the connection between the first and second phases, which allows us to use several different techniques to gather traces, and several different offline algorithms. The trace contains, at a minimum, a *(timestamp, sender, receiver)* tuple for each message, but might include some additional information. Because the information we need depends on which algorithm is used, we will describe the algorithms before describing the specifics of gathering traces.

5. ALGORITHMS

Our key technical, as opposed to logistical, challenge is to infer causal path patterns and latencies from relatively simple message traces. We have developed two distinctly different algorithms. One depends on the use of RPC-style communication, and operates on individual messages in the trace. The other is able to handle free-form message-based communication, and uses signal-processing techniques to extract causal information from traces.

5.1 The "nesting algorithm"

The first algorithm combines all of the per-edge traces into a single global trace and explicitly examines the individual trace entries to infer how calls are "nested." That is, if node A calls node B and then B calls C before returning to A, the B-to-C call is *nested* within the A-to-B call. We call this the "nesting algorithm." It is roughly linear (in time and space) in the size of the trace. However, it requires RPC-style communication (in which the messages are calls and returns) to make its inferences.

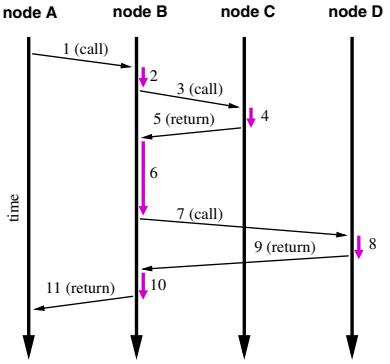


Figure 3: Timelines for example of Figure 2

Figure 3 illustrates the nesting property for the example causal path from Figure 2. This figure shows the causal connections between the timelines for each of the four nodes. In this example, the call from B to C (messages 3 and 5) is nested in the call from A to B (messages 1 and 11). The algorithm infers nesting relationships by examining the timestamps of the messages in the trace.

While any given pair of calls might appear to be nested purely by accident, if the same nesting relationship appears repeatedly in a lengthy trace then we can infer with high probability that the nesting represents a causal relationship between these calls. The processing latencies in nodes B, C, and D may be calculated directly from the message timestamps. We build multi-node call paths based on apparent causal relationships, then apply simple statistical methods to infer which paths are both non-accidental and significant contributors to overall system latency.

5.1.1 Details of the nesting algorithm

A *call pair* is a tuple describing a single call from one node to another and its matching return. It contains the timestamps of the two messages and the names of the nodes. The nesting algorithm consists of four steps:

1. Find call pairs in the trace.
2. Find all possible nestings of one call pair in another, and estimate the likelihood of each candidate nesting.
3. Pick the most likely candidate for the causing call for each call pair.
4. Derive call paths from the causal relationships.

We first illustrate the algorithm using the example in Figure 3. Step (1) groups trace entries 1 and 11—the call $A \rightarrow B$ and the return $B \rightarrow A$—into call pair $(A, B, 1, 11)$; entries 3 and 5 into call pair $(B, C, 3, 5)$; and entries 7 and 9 into call pair $(B, D, 7, 9)$. (For ease of explanation, in this example we use the message numbers as the timestamp values.)

Step (2) examines each call pair to determine the set of calls that might have caused it. Here, $(B, C, 3, 5)$ and $(B, D, 7, 9)$ both occur between the beginning and end of $(A, B, 1, 11)$. $(A, B, 1, 11)$ is the only call that encloses $(B, C, 3, 5)$ and $(B, D, 7, 9)$. In a more complex example, a call pair might be nested within several different "parent" calls, which would have to be ranked by likelihood.

Step (3) chooses the most likely parent call for each call pair in the trace as its causal parent, based on aggregate information from all other call pairs between the same nodes. Step (4) again examines each call pair and creates a call path starting from each call pair that was not nested in any other call pair. Since $(A, B, 1, 11)$ is the parent for two call pairs, it creates the path $A \rightarrow B \rightarrow C; D$. The call pairs $(B, C, 3, 5)$ and $(B, C, 7, 9)$ do not initiate paths because they are nested in $(A, B, 1, 11)$.

We store all path patterns in a table. If a new path matches a path pattern already in the table, then the existing pattern is updated with the latencies for the new path. Otherwise, a new path pattern is initialized with the path's latencies. At the end of the algorithm, the path patterns can be sorted by their frequency and all path patterns can be displayed with average latencies or latency distributions for each node.

Figure 4 shows pseudo-code for the four steps of the nesting algorithm, which we now explain in more detail.

5.1.1.1 *Identifying call pairs.*

This part of the algorithm (procedure FindCallPairs in Figure 4) matches call and return trace entries into call pairs using a hash table $T_{opencalls}$ which is indexed by (sender, receiver, callid) tuples. Call identifiers are based on packet header information and are used only to match calls with returns; they need not be end-to-end request identifiers as used in Magpie or Pinpoint. Messages that do not have a matching call or return message are discarded during this step; noise in the trace—extraneous and dropped messages—does not impact the rest of the algorithm.

When call identifiers are not provided or are not unique (for example, when RPC packets are retransmitted), the entries for a given (sender, receiver, callid) are sorted by timestamp. If multiple calls occur from $A \rightarrow B$ before any returns from $B \rightarrow A$ then each return is matched with the earliest unmatched call. This heuristic works when no call path progresses faster than its predecessors, but fails otherwise. For example, given the correct call pairs (A, B, i, j) and (A, B, k, l), if $i < k$ but $j > l$ then the algorithm will incorrectly create the call pairs (A, B, i, l) and (A, B, k, j). More importantly, this heuristic cannot handle extraneous or dropped messages. However, we believe that we can find usable call identifiers in message headers in most cases, and so we have ignored the problem so far.

This step also identifies all possible parents for each call pair. At the time the return message of the call pair (B, C) is processed, we find all call pairs $(-, B)$ in $T_{opencalls}$ with an earlier call timestamp. (B, C) is nested inside all of them.

5.1.1.2 *Scoring potentially-causal nestings.*

A call pair (B, C, k, l) might be nested in many (A, B, i, j) call pairs, but it is only directly caused by one such parent. The ScoreNestings procedure estimates the likelihood that each nesting relationship is really a causal relationship. We do this using a *scoreboard* that records the prevalence, in the entire trace, of the delays between the two call messages in a potentially-causal nesting.

```
1.  procedure FindCallPairs
2.  for each trace entry (t1 , CALL/RET, sender A, receiver B, callid id)
3.    case CALL:
4.      store (t1 ,CALL,A,B,id) in T_opencalls
5.    case RETURN:
6.      find matching entry (t2, CALL, B, A, id) in T_opencalls
7.      if match is found then
8.        remove entry from T_opencalls
9.        update entry with return message timestamp t2
10.       add entry to T_callpairs
11.       entry.parents := {all callpairs (t3, CALL, X, A, id2)
12.         in T_opencalls with t3 < t2}

13. procedure ScoreNestings
14. for each child (B, C, t2, t3) in T_callpairs
15.   for each parent (A, B, t1 , t4) in child.parents
16.     scoreboard[A, B, C, t2 − t1 ] += (1/|child.parents|)

17. procedure FindNestedPairs
18. for each child (B, C, t2, t3) in call pairs
19.   maxscore := 0
20.   for each p (A, B, t1 , t4) in child.parents
21.     penalty = /* see Section 5.1.1.3 */
22.     score[p] := scoreboard[A, B, C, t2 − t1 ]·penalty
23.     if (score[p] > maxscore) then
24.       maxscore := score[p]
25.       parent := p
26.   parent.children := parent.children ∪ { child }

27. procedure FindCallPaths
28. initialize hash table T_paths
29. for each callpair (A, B, t1 , t2)
30.   if callpair.parents = ∅ then
31.     root := new path starting at A
32.     root.edges := { CreatePathNode(callpair, t1 ) }
33.     if root is in T_paths then update its latencies
34.     else add root to T_paths

35. function CreatePathNode(callpair (A, B, t1 , t4), tp)
36.   node := new node with name B
37.   node.latency := t4 − t1
38.   node.call_delay := t1 − tp
39.   for each child in callpair.children
40.     node.edges := node.edges ∪ { CreatePathNode(child, t1 ) }
41.   return node
```

Figure 4: Pseudo-code for the nesting algorithm

The scoreboard represents the set of all nesting-delay tuples $(A, B, C, delta)$, where $delta$ is the time difference between the call from A to B and the subsequent call from B to C; each tuple has an associated value. The scoreboard entries for a given nesting thus form a histogram of these delay values. However, each increment to a histogram count is weighted by the number of possible parent calls: if there are N possible parent calls for a given child call, then the scoreboard value for each of these N tuples is incremented by $1/N$.

We actually store each histogram as a set of exponentially-sized bins, efficiently representing the large range of delay values that might appear in real traces. We find that 340 bins (indexing the histogram by $\log_{1.05} delta$) gives reasonably accurate results for intervals between 1 msec. and 2 hours. The number of histograms is equal to the number of (A, B, C) triples such that a call $B \rightarrow C$ is nested in a call $A \rightarrow B$ at least once. This number, which is independent of trace length, is at most n^3, for n nodes; in practice it should be significantly lower.

After scoring all of the call pairs, we optionally smooth the histograms by convolving them with a Gaussian normal curve. Smooth-

ing helps accuracy when there is skew in the message timestamps, as shown in Section 7.5.6; it has little effect in traces without skew.

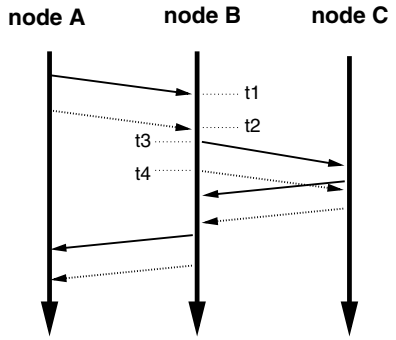


Figure 5: Example of parallel calls

Figure 5 shows an example in which two $B \rightarrow C$ calls are each nested in two $A \rightarrow B$ calls, creating four possible sets of parent-child pairings. However, the "medium-length" delay ($t_3 - t_1$ and $t_4 - t_2$) occurs twice as often as the "long" delay ($t_4 - t_1$) or the "short" delay ($t_3 - t_2$). Thus, the histogram for (A, B, C) has a peak at the medium-length delay.

5.1.1.3 Choosing unique parents.

The FindNestedPairs procedure chooses the most likely causal parent for each call pair. The inference that any given nesting is a causal relationship is based on the scoreboard generated in the previous step, combined with simple heuristics about the parent's calls to other possible children. For each call pair (B, C, t_2, t_3) in the trace, we consider each possible parent (A, B, t_1, t_4) and generate a score for the relationship. The raw score is simply the value of $(A, B, C, t_2 - t_1)$ in the scoreboard. The raw score is then scaled using three *penalties*:

- Overlapping-child penalty: We count the number of children $c_{overlap}$ already assigned to the given parent that overlap in time with the current call pair, and multiply the score by ${c_{overlap}}^{-x}$.
- Same-child penalty: We count the number of children c_{same} already assigned to the given parent that have the same destination as the current call pair, and multiply the score by ${c_{same}}^{-y}$.
- Generic-child penalty: We count the number of children c_{any} already assigned to the given parent, and multiply the score by ${c_{any}}^{-z}$.

The parameters x, y, and z are configurable. In our experiments, we get the most predictable, near-optimal performance across all workloads with $x \approx 2$ and $y = z = 0$. However, there are individual workloads for which different values perform better.

In Figure 5, each $B \rightarrow C$ child call has two possible $A \rightarrow B$ parents, but each child has one parent for which the scoreboard includes a peak at the medium-length delay ($t_3 - t_1$ and $t_4 - t_2$). Based on this inference, FindNestedPairs assigns each $B \rightarrow C$ child call pair to one of the $A \rightarrow B$ parent call pairs, as shown with the solid and dashed lines in the figure. The overlapping-child penalty encourages FindNestedPairs to assign the two children to different parents. Tie scores when considering parents for a given child are broken by assigning the child to the earliest tied parent.

5.1.1.4 Creating and aggregating call paths.

The final step, FindCallPaths, coalesces the causal relationships found in step (3) into call paths, and keeps aggregate latency statistics for each path pattern. We use hash table T_{paths} to find path patterns quickly.

The *latency* of a node is the total time spent in processing at that node, including at any nodes that it calls. The *call_delay* of a node is computed as the time between the call to its parent and the inferred causally-related call to this node.

5.1.2 Time and space complexity

Finding call pairs is linear in both time and space in the size of the trace: each trace entry is examined once and put into one call pair. Finding nested call pairs is linear in both time and space in the total number of nesting relationships. This number is the product of the number of trace entries and the mean per-node parallelism during the trace. We define per-node parallelism as the average number of candidate parents for each child (see Section 5.1.1.3). Creating and aggregating call paths is linear in the number of messages in the trace: each message either begins a new call path or belongs to exactly one existing call path. Overall, the algorithm is linear in the number of messages times the mean per-node parallelism.

5.2 The "convolution algorithm"

Unlike the nesting algorithm, our second algorithm finds causal relationships by considering the aggregation of multiple messages, rather than by examining messages individually. The algorithm separates a whole-system trace into a set of per-edge traces, and treats each of the per-edge traces as a time signal. The central idea of the algorithm is to convert traces into time signals and then use signal processing techniques to find the cross correlations between signals. It considers the trace of messages from A to B separately from the trace of messages from B to A, so this algorithm can be used on traces of free-form message-based communication, not just RPC-style traces.

The results of this algorithm are directed graphs, in which a node might appear several times (e.g., $A \rightarrow B \rightarrow A \rightarrow C$). To avoid confusion between the graph of the distributed system itself and the output graph, we use the term *vertex* for the graph vertices and *node* for the components of the system.

Figure 6 summarizes the algorithm using pseudo-code. Given a root node i and a message trace T, the algorithm first creates a vertex x_i in the output graph. Then it considers the messages with source i: for each different destination node j in those messages, there is a causal relationship between i and j, so the algorithm creates a vertex x_j and adds an edge from x_i to x_j.

The algorithm then continues the path from j by calling ProcessNode. Procedure ProcessNode calls FindCausedMessages to find the sets of messages with source j that appear to be caused by the messages from i to j. Each set contains messages with a single destination node k and a common delay d: the set indicates that a message from j to k was sent exactly d time units after a message from i to j. For each set, it adds a vertex x_k with label k and edge (x_j, x_k) with label d to the graph, and recursively continues along the path from k (i.e., it creates the graph in depth-first order).

Function FindCausedMessages is the heart of the algorithm. It computes the causal delays d, which are *time shifts* between the messages arriving at j and the messages leaving j. To find these time shifts, it converts the messages V from i to j into an indicator function $s_1(t)$. This function is defined to be

$$s_1(t) = \begin{array}{l} 1 \text{ if } V \text{ has a message in time interval } [t-\epsilon, t+\epsilon] \\ 0 \text{ otherwise} \end{array}$$

where ϵ is a small fixed constant and $[t-\epsilon, t+\epsilon]$ is a short closed interval. It similarly converts all messages sent from node j into an indicator function $s_2(t)$. It then computes the *cross correlation* $C(t)$ of $s_2(t)$ and $s_1(t)$. $C(t)$ is defined to be the *convolution*

```
1.  procedure FindPathsFromRoot(i)
2.  T_i := trace of messages with source i
3.  output_graph := graph with one vertex x_i labeled i
4.  for each destination node j in T_i do
5.      V := messages in T_i with destination j
6.      add vertex x_j labeled j and edge (x_i, x_j)
7.          to output_graph
8.      ProcessNode(j, x_j, V)

9.  procedure ProcessNode(j, x_j, V)
10.     T_j := trace of messages with source j
11.     O_1, ..., O_m := FindCausedMessages(V, T_j)
12.     for n := 1 to m do
13.         k := O_n.node;  W := O_n.messages
14.         d := O_n.delay
15.         add vertex x_k labeled k and edge (x_j, x_k)
                labeled (|W|, d) to output_graph
16.         ProcessNode(k, x_k, W)

17. function FindCausedMessages(V, Z)
18.     m := 0
19.     C := FindCorrelation(V, Z)
20.     find positions of spikes of C(t)
21.     for each spike position d found do
22.         Z' := messages in Z with a timestamp equal
                to some timestamp in V shifted by d ± v
23.         for each destination node k in Z' do
24.             m := m + 1
25.             O_m.node := k;  O_m.delay := d
26.             O_m.messages :=messages to k in Z'
27.     return O_1, O_2, ..., O_m

28. function FindCorrelation(V, Z)
29.     s_1 := indicator function for V
30.     s_2 := indicator function for Z
31.     return cross_correlation(s_2, s_1)
```

Figure 6: Pseudo-code for the convolution algorithm

of s_2 and the time inverse of s_1[1], which is why we call this the "convolution algorithm." Roughly speaking, $C(t)$ has a spike at position d if and only if $s_2(t)$ contains a copy of $s_1(t)$ time-shifted by d. Figure 7 shows the convolution for an example $s_1(t)$ and $s_2(t)$.

To detect the spikes, if any, in $C(t)$, we compute the mean and standard deviations of C. We consider a point to be a "spike" if it is a local maximum N standard deviations above the mean, where the parameter N is a small number (e.g., 4). There may be many such local maxima close together. Rather than consider each one to be a separate spike, we require at least one point that is less than S standard deviations above the mean between spikes, where $S < N$ is another small number (e.g., 3). Among the candidate points for a given spike, we choose the largest to represent the spike.

5.2.1 *Discretization of the indicator function*

The definition for $s_1(t)$ assumes that t is a continuous time parameter. In practice, we need to discretize time. To do so, we choose a time quantum μ and then treat t as an integer multiple of μ. The definition of $s_1(t)$ is then modified as follows:

$s_1(t) =$ square root of number of messages in V during time interval $[t\mu, (t+1)\mu)$, where t is an integer.

Several discretizations are possible but the above definition produces the most accurate results and is what we implemented. Note that $s_1(t)$ can be represented by an array. When there are time quanta with lots of messages, if $s_1(t) = x$ and $s_2(t+d) = x$ then the (discrete) convolution of $s_2(t)$ and $s_1(-t)$ at position d includes an x^2 term. The square root in the definition compensates for this square. We similarly change the definition of $s_2(t)$.

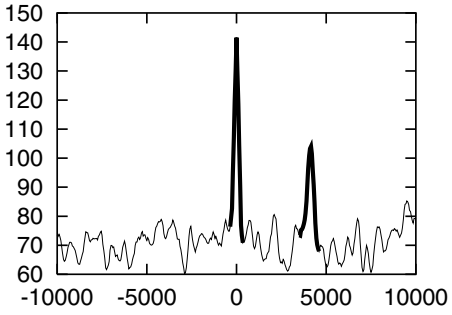


Figure 7: Example of convolution output, showing two spikes with bold lines. The x-axis represents the time shift; the y-axis roughly estimates the number of messages matching a given shift.

5.2.2 *Dealing with delay variances*

The algorithm described so far performs best when the node delays have little variance. For example, for each message from A to B, B sends another message to C after the same fixed delay. When the variance is significant, we can get better results by increasing the parameter v in line 22 of the algorithm, to allow delay variations of that magnitude.

5.2.3 *Dealing with undesirable paths*

Further improvements to the convolution algorithm remove "noise" (low-frequency paths), suppress the detection of accidentally short paths and of cycles, and suppress edges with negative apparent delay. Space limitations prevent us from describing them here.

5.2.4 *Other improvements*

Our implementation of the convolution algorithm includes numerous other features to improve accuracy. For lack of space we do not describe these improvements, but the measurements we present in Section 7 reflect their effects.

5.2.5 *Time and space complexity*

The convolution algorithm must store the m messages in the trace, and the vectors containing discretized indicator functions. At any time, there is a constant number of such vectors. The size of each vector is bounded by $S = T/\mu$, where T is the duration of the longest trace and μ is the time quantum. Hence, the overall space complexity is $O(m+S)$.

The time complexity of the algorithm is proportional to the time to traverse the trace and the time to compute convolutions of discretized indicator functions. Convolutions of vectors of size S can be computed in time $O(S \log S)$ using fast fourier transforms. The number of times the trace is traversed and a convolution is computed is proportional to the number e of edges in the output graph

[1]The convolution of two functions $f(t)$ and $g(t)$ is another function, denoted $f \otimes g(t)$, defined by $f \otimes g(t) = \int_{-\infty}^{+\infty} f(u)g(t-u)du$. The discrete version of this definition is $(f \otimes g)_i = \sum_{j=-\infty}^{+\infty} f_j g_{i-j}$.

G. Hence, the overall time complexity is $O(em + eS \log S)$. In practice, we find that the second factor, $eS \log S$, tends to dominate the first.

5.3 Comparison of the two algorithms

Our two algorithms have different strengths and weaknesses. Often these strengths are complementary: sometimes one algorithm works better, sometimes the other. Here we contrast the algorithms in terms of their utility.

5.3.1 RPC vs. free-form messages

The nesting algorithm explicitly works only with systems that use RPC-style communication. The convolution algorithm can find causal relationships in any form of message-based system. The limited applicability of the nesting algorithm is not without benefits, though: because it "knows" that a system is RPC-based, it provides a more concise representation of such systems than could the convolution algorithm.

Some common forms of RPC-based systems pose a problem for the nesting algorithm as we have implemented it, and currently can only be analyzed with the convolution algorithm. If a system forwards RPC calls or returns asymmetrically (e.g., A calls B, B forwards the call to C, and C replies directly to A) then we fail to detect this as a single RPC call. Also, if a called node replies to a call *before* issuing a causally-related subsequent call, there is no obvious nesting relationship between the two calls. (This can happen, for example, when an intermediate node uses delayed write-back caching.) We believe that the nesting algorithm can be expanded to deal with these cases, but doing so is future work.

On the other hand, the convolution algorithm has some drawbacks with RPC-style path patterns. Given a path pattern $A \rightarrow B \rightarrow C \rightarrow B \rightarrow A$, the algorithm will not only report this path, but also $A \rightarrow B \rightarrow A$. This is because there is a causal relation between $A \rightarrow B$ and $B \rightarrow A$. If a node appears many times on a path, the algorithm will report a large number of derived paths that are not very interesting. We believe it is possible to automatically filter out such paths, while preserving legitimate paths, by using frequency counts; this is future work. The nesting algorithm correctly finds the right number of instances of each pattern, as we show in Section 7.5.2.

5.3.2 Rare events

The convolution algorithm looks for spikes in the cross correlation of two signals. Therefore, it cannot be used to search for rare events, especially those with high delay variance.

The nesting algorithm explicitly analyzes every RPC message for its relationship with other messages, and therefore can find rare events. However, distinguishing the rare events of interest from the more frequent but uninteresting events is still an unsolved problem. Also, the scoreboard mechanism described in Section 5.1 currently biases the algorithm away from rare events: they will be found most easily when there are few overlapping calls among the same nodes.

5.3.3 Detail required in traces

Our tools would ideally require no information about message formats. In practice, this goal means that the algorithms should use only information available from widely deployed standards with self-describing formats. The convolution algorithm effectively meets this ideal; it requires only timestamps and sender and receiver identifiers.

The nesting algorithm further requires that trace entries be marked as either RPC calls or returns. (In a few cases, this information can be inferred based on *a priori* knowledge of address formats, such as UDP's "well-known" port numbers.) The algorithm also performs much better if the trace system can extract call identifiers from the RPC messages.

5.3.4 Time and space complexity

As discussed in Section 5.1.2, the nesting algorithm runs in time and space linear in the number of traced messages times the amount of parallelism in the trace. Generally, the trace length (in messages) dominates. As we will show in Section 7.6, practical running times are quite low—much lower than the duration of the traces themselves—and the space overhead is more likely to be the limiting factor.

The convolution algorithm, as discussed in Section 5.2.5, has space complexity linear in the length of the trace (measured either by message count or total number of time quanta, whichever is larger), with a modest constant factor. Running time is the dominant cost for the convolution algorithm; as we show in Section 7.6, it can be much slower than the nesting algorithm. In practice, there is a tradeoff between increased precision of the delay results (decreased μ) and longer running time.

5.3.5 Visualization

The two algorithms provide different visualizations, even when applied to the same trace. For RPC-based systems, the nesting algorithm provides a more compact output, because the convolution algorithm does not combine calls and returns into one graph edge.

5.4 Visualization of results

The visualizations we show in this paper are rather primitive, in the form of graphs produced by the "dot" program [8]. We use similar but not identical formats for the output of both the nesting and convolution algorithms.

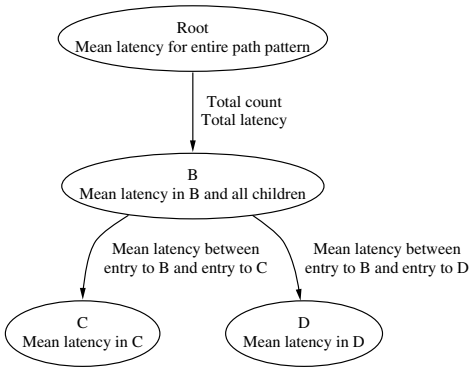


Figure 8: Output format for nesting algorithm

The output of the nesting algorithm, as depicted in Figure 8, is a graph showing the procedure call hierarchy for a specific causal path pattern. The total number of instances for the pattern, and the total latency for this pattern, are shown next to the first edge in the graph. A node's ellipse includes its name and the mean latency for all activity in the node and its children. An arrow representing the direction of a call is labeled with the mean latency between entry into the parent node and entry into the child node.

The nesting algorithm actually computes the full distribution for each latency, rather than just the mean. We show only the mean in our "dot" visualizations, to avoid cluttering the output. We have started developing an interactive visualization tool that provides a richer display, including delay distributions. This tool also allows the user to sort paths by frequency or total latency, and highlights the individual nodes that contribute the most latency.

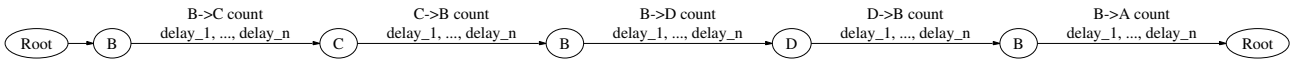


Figure 9: Output format for convolution algorithm

With the representation of Figure 8, internal delays as shown in Figure 2 may be calculated from the node and edge labels. Our interactive tool simultaneously displays both the tree-structured view of a path and a timeline view, which makes the internal delays explicit and clearly shows the parallelism between calls.

The nesting algorithm inherently generates trees whenever a node calls more than one child. The convolution algorithm views every path as a linear sequence of nodes, sometimes with multiple visits to a node. Figure 9 shows the output format for the convolution algorithm given the same RPC-style system depicted in Figure 8. Each directed edge is labeled with a set of delays, representing the time(s) that the preceding node spends before sending a message to the next node (i.e., the shift(s) found for the spike(s)). The delays are ordered by declining frequency. Each edge is also labeled with the total count of messages with those delays.

6. OBTAINING TRACES

Our approach depends on tracing all (or nearly all) of the messages between nodes in the distributed system. This requirement leads to numerous challenges. This section describes the techniques we are developing to obtain traces; we discuss specific trace sets in Section 7.1.

6.1 General concerns

Our black-box assumptions simplify the tracing problem, because we need relatively little information about each message. The convolution algorithm needs just the timestamp, sender, and receiver. The nesting algorithm also needs to know whether messages are calls or returns, and can benefit from call identifier information (e.g., from RPC headers), which improves the accuracy of call-pair matching. Therefore, we need not parse the messages too deeply into the protocol stack.

We might need rather large trace sets to analyze certain distributed systems. The message rate might be quite high and the trace duration necessary for revealing interesting call path patterns might be long. While handling large trace sets creates logistical challenges, the problem remains feasible because large traces stress aspects of computer systems that scale well: local area network bandwidth, and storage bandwidth and capacity.

While a suitable trace collection and analysis system might represent a significant capital cost above that of the system under test, we intend this hardware for use during debugging. It is appropriate to invest in debugging equipment that can be re-used for various systems under development, especially if this investment increases programmer productivity.

The term "black box" can be applied with more or less rigor, depending on the granularity of the nodes of interest and on how hard it is to extract the minimal message information that we need. For example, the developers of the systems listed in Section 3.1 use "black box" to mean "application-code generic." We aspire to a more rigorous black-box ideal, a tool that requires absolutely no support from the nodes of the system, requires no message-specific knowledge beyond widely-deployed standards, and does not perturb system performance at all.

Passive network tracing can approximate this ideal, but cannot always expose the nodes at the appropriate levels of granularity. If the nodes of interest are, for example, processes or J2EE objects, we must obtain traces more intrusively. Non-passive tracing compromises our zero-knowledge, zero-instrumentation, and zero-perturbation goals, but if the costs can be minimized then our tools are still useful. Our approach also has the advantage, over systems (such as in Section 3.1) requiring infrastructural changes, that we can merge traces from both passive monitoring and more intrusive monitoring to get a unified view of a complex system built from "legacy" components.

We are developing techniques to obtain traces at various layers of a system, and with varying levels of intrusiveness. These include passive network monitoring, middleware instrumentation, kernel instrumentation, and (in certain cases) application instrumentation. We describe our specific approaches in the following sections.

6.2 Passive network tracing

When the nodes communicate via a network, we can obtain message traces through passive network tracing (or "packet sniffing"). Passive tracing, at least in principle, does not perturb the system under test, and requires no software changes to the system. This enables its use in risk-averse production environments and on legacy systems. Passive tracing is therefore our preferred mode.

However, while we have successfully collected and analyzed passive traces, none of the experiments reported in this paper are based on passive traces, so (given space constraints) we will only briefly discuss the issues associated with passive tracing.

A packet trace requires some processing to be useful for our analysis tools. Problems include identifying nodes based on addresses at various protocol levels; finding message boundaries when messages span packets or start in the middle of packets; and identifying calls and returns, and extracting call identifiers for RPC protocols. These are not novel challenges; many researchers and commercial products have done elaborate analysis based recovering or inferring high-level information from raw packet traces [7, 21].

6.2.1 Mechanics of passive tracing

With older broadcast-bus LANs, it was easy to passively capture all packets from one monitoring point. Modern switched LANs make the problem harder. We see two possible approaches:

Port mirroring , which is supported by many switch vendors, allows a switch to be configured to copy ("mirror") some or all packets to a dedicated monitoring port. It allows us to treat application hosts entirely as black boxes (we need not install any software on those hosts) and should not perturb the system under test.

Packet sniffing at each participant host applies when the hosts support programs such as tcpdump [14]. After trace capture, the traces are merged in post-processing (see Section 6.5).

High packet rates can overload a sniffing system or its incoming link, because we cannot flow-control the messages to avoid this. The scalability of our approach depends somewhat on this issue, although our algorithms tolerate some packet loss (see Section 7.5.5).

Researchers at the University of Waikato and Endace Technology [17] have achieved a capture rate of close to 20M packets/sec. using a commodity dual-CPU server and special-purpose network capture cards. We experimented with a relatively small server (AlphaServer DS10, 618 MHz, Tru64 UNIX V5.1A) running tcp-

dump, and found that it could capture slightly over 25,000 packets per second (albeit with some losses).

6.3 Tracing in a J2EE system

Many modern distributed systems are built on J2EE [25], using Enterprise Java Beans (EJBs) to represent components. Members of the Pinpoint project (see Section 3.3) have tools to trace inter-EJB calls and returns, and they graciously shared their code with us. Their tracing system [16] tags all messages on a call path with a single end-to-end request-ID, but we can ignore the end-to-end information, to test our algorithms as if we only had simpler traces.

J2EE-level tracing imposes runtime costs and perturbs system performance. The Pinpoint tools are not optimized for our purposes, and might never be cheap enough to run full-time in a production environment. However, our tools are meant for performance debugging, not system management, so tracing need not be enabled full-time. A system owner can enable tracing only during a debugging phase, exactly the time when the owner is willing to pay the price of some extra short-term overhead in the interest of solving a long-term problem.

6.4 Application-level tracing

Our black-box approach does not normally involve modifying applications to generate message traces directly, but we are not too proud to use such traces when available. Some applications already generate, in normal operation, sufficient tracing or logging information for our purposes (perhaps with some post-processing).

6.5 Merging traces

We might need to merge traces collected at different points in the distributed system (e.g., packet sniffers at multiple hosts) or at different layers (e.g., both packet sniffing and J2EE tracing).

We simplify the trace-merging process by adopting a uniform representation for traces, for example:

timestamp	*operation*	*sender*	*rcvr*	*ID*
1047680084.482205	CALL_SENT	nodeA	nodeB	id37
1047680084.483575	RET_SENT	nodeB	nodeA	id37

We then merge the individual trace entries in timestamp order. Both algorithms described in Section 5 can handle minor clock skews, although synchronized clocks improve our accuracy. Mills [20] has shown that the widely-deployed NTP protocol can synchronize clocks on the same LAN with an RMS error of under 1 msec., and over the global Internet "usually less than 5 ms." This accuracy is usually more than sufficient, because trace timestamp resolution seldom is as low as 1 msec.

This leaves several problems, such as duplicate entries (e.g., from sniffing packets at both ends of a link) and node-naming inconsistencies between traces made at different levels. We have developed techniques to solve several such problems, but we have not yet tested them adequately.

7. EXPERIMENTS AND RESULTS

In this section, we describe several experiments that show how our tools might be used in practice. We also describe experiments to validate the accuracy of our tools.

7.1 Trace sets

We would like to test our tools on traces from a heavily-used "real-world" application, such as a multi-tier Web server. However, access to such systems is tightly controlled, and we have not yet succeeded in obtaining the necessary traces.

Therefore, in order to debug and test our algorithms and trace-collection techniques, we have obtained several traces of varying degrees of realism. Here we describe the trace sets and how they were obtained; subsequent sections describe what we learned from each trace set.

Note that these traces were *not* collected using purely black-box techniques. Rather, we have chosen traces that can demonstrate the accuracy of our algorithms: by starting with "white box" traces and then converting them to a "black box" form (i.e., by removing information) we are able to explicitly evaluate how well the algorithms work (see Section 7.5).

7.1.1 Trace generator

In order to test our algorithms on specific cases, including trace scenarios we expect to be challenging, we wrote "maketrace," a *tracelet-based* trace generator. A *tracelet* is a template for an ordered sequence of messages between nodes, with parameterized Gaussian delay between messages. A tracelet can represent a specific causal path pattern through a distributed system; it can also represent an explicit interleaving of several causal paths, if we want to test how well our algorithms disentangle such an interleaving.

Maketrace takes a configuration file that specifies a set of tracelets, and for each tracelet a parameterized uniformly random delay between sequential invocations (representing a client's "think time"). The configuration also specifies how many instances of each tracelet sequence run in parallel. Maketrace thus directly constructs arbitrarily long traces by instantiating tracelets, rather than by generating traces as a side-effect of simulating a distributed system.

7.1.2 J2EE traces

Our J2EE traces consist of inter-EJB calls in the PetStore v.1.3.1 example application [24], running on a single-node JBoss v.3.0.6 server [15], on a 2-CPU 1GHz Pentium III with Linux 2.4.9. A load generator ran on the same host, emulating 24 clients with several workload profiles and a mean inter-request think time of 7 seconds.

We obtained two traces, each about three hours long and including about 1.3 million messages. (Each inter-component call results in two messages.) In one trial, we artificially increased the delay in one leaf component, and were able to find the added delay easily using both of our algorithms. However, for most of the experiments reported in this section, we used a 2000-second prefix of each trace; this avoids excessive run times for the convolution algorithm (see Section 7.6).

7.1.3 Received-header trace

While searching for a large, real-world application that is not primarily RPC-based, we realized that email transit service is ideally suited to testing the convolution algorithm. Because most email messages pass through several servers, and almost all servers add "Received" headers (with source, destination, and timestamp) to each message, we can extract these Received headers and treat them individually as entries in a trace of inter-node message transmissions.

(We emphasize that this is *not* the best way to use Received headers for causal-path analysis of an email system. By treating the Received headers of a given message as separate trace entries, rather than directly extracting the path the message had followed, one creates an unnecessarily hard problem. However, this problem is exactly the one that the convolution algorithm is meant to solve, so it *is* a good test of our approach.)

One of the authors logged all of his incoming message headers for this experiment. He gets lots of email (partly because he has several email addresses that resolve to the same mailbox, and lots of spam targets more than one of these addresses). Over two months,

he received 11,683 email messages including a total of 81,044 usable Received headers. A small number of Received headers were excluded because of unparseable timestamps, or because they were from generic hostnames, such as "localhost" or "unknown," that would have created false connections between many paths. We also excluded all forwarding hops outside the corporate email system, to avoid an explosion in the number of paths.

7.1.4 *Other traces*

In addition to the traces described above, we have applied our algorithms to several other traces from real systems. These include one gathered from a distributed file system, and another from instrumented inter-method communication in an embedded system. We were able to find the correct causal paths in these traces. We do not describe these further, both for space considerations and because the results do not illustrate any novel issues other than those revealed by the traces described above.

7.2 Results: Tracelet-based multi-tier traces

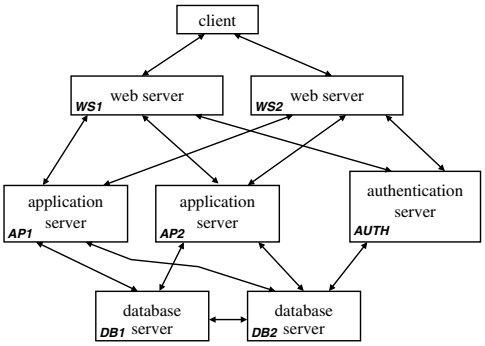


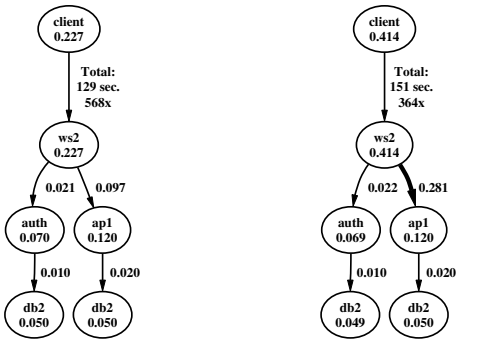
Figure 10: Multi-tier configuration (simplified version of Figure 1)

We used maketrace to generate a variety of traces simulating the multi-tier configuration shown in Figure 10. Some of these traces have an additional 200 msec. delay inserted at node *WS2*, between the serial calls to *AUTH* and to either *AP1* or *AP2*, so that we can test if our algorithms correctly measure such delays. We refer to these as added-delay traces.

To test the nesting algorithm, we generated normal and added-delay traces including about 200,000 messages each. Figure 11 shows the results for the normal case, with the most frequent causal path patterns ranked left to right in order of declining total latency. While this figure is too dense to depict the particulars of any specific pattern, it shows how a load-balancing configuration, such as in Figure 10, can generate an exponential increase in the set of paths. In effect, there are only a few abstract paths in this figure, and a good visualization tool (future work) would cluster together isomorphic graphs with similar counts and delays.

We then ran the nesting algorithm on the added-delay trace. Figure 12 shows the "normal" and "added-delay" output for one specific causal path pattern that includes the *WS2* node. One can easily infer from the delays on the graph edges (especially the edge between *WS2* and *AP1*) that there is approximately 200 msec. of added delay in Figure 12(b), albeit slightly underestimated by the algorithm.

We analyzed the same traces with the convolution algorithm ($\mu = 0.1$). Figure 13 shows the results, for the same causal path pattern as in Figure 12. This algorithm generates long paths for RPC-style call patterns, because it looks at the call and return messages independently, rather than as unified RPCs. However, it is



(a) Normal trace (b) Added-delay trace

Figure 12: Multi-tier results from the nesting algorithm

quite good at assigning the blame to the correct node (marked on the edge between *WS2* and *AP1*, in bold), and at correctly measuring the extra delay.

7.3 Results: J2EE traces

We ran two traces of the PetStore system: one with no added delay, the other with a constant 50 msec. delay added in each call of the */mylist.jsp* node.

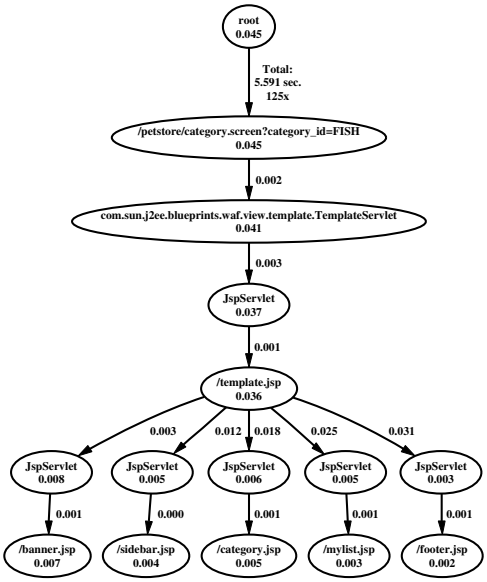


Figure 14: PetStore results, normal configuration (nesting algorithm)

Figure 14 shows one frequently-invoked causal path pattern found by the nesting algorithm, although not the most frequent one. (The convolution algorithm produces similar results.) Figure 15 shows the same path when excess delay is inserted in node */mylist.jsp*, shown in gray. One can clearly see the excess, when comparing this diagram to Figure 14, not only at the slow node, but also in its parents and in the totals for the entire path. (The excess appears to be slightly larger than the nominal 50 msec. delay added; this might be an artifact of Linux's 10 msec clock granularity.)

7.4 Results: Received-header trace

We ran the convolution algorithm for the Received-header (email-header) trace. With a quantum of 30 sec., the algorithm

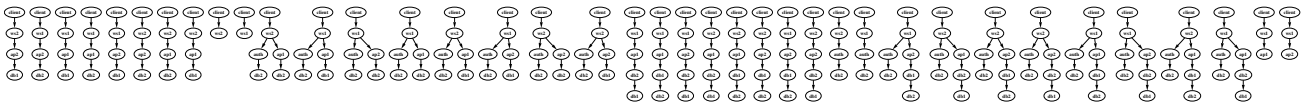


Figure 11: Expanded multi-tier results from the nesting algorithm (you are not expected to be able to read this)

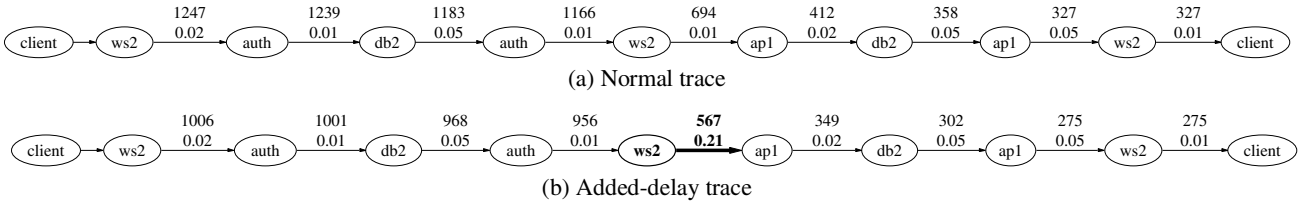


Figure 13: Multi-tier results from convolution algorithm

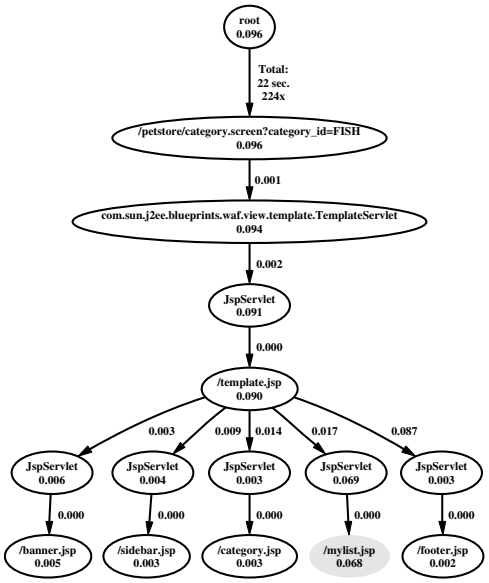


Figure 15: PetStore results, constant-delay config. (nesting algorithm)

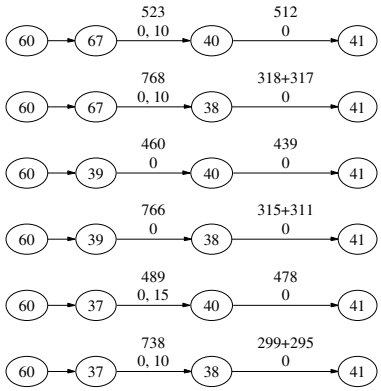
reports all delays as zero, implying real delays between zero and 29 secs. As the quantum is decreased to 5 sec., the algorithm reports some secondary spikes above zero secs.; Figure 16 shows the most frequent paths from this trial. Node "names" in this figure are arbitrary integers. Note that the primary spikes on all paths are at zero, because most of the time messages are forwarded immediately. However, some paths show secondary spikes at 10 or 15 seconds. We verified from the original trace that these spikes are accurate.

7.5 Results: Validation of accuracy

So far we have discussed our results primarily in qualitative terms. Here we attempt to quantify the accuracy of our algorithms.

7.5.1 *Metrics for evaluating accuracy*

To evaluate the accuracy of our algorithms, we developed a set of simple metrics that quantify the discrepancies between the "ground truth" of a trace (the actual call paths traversed during the trace) and the call paths inferred by one of our algorithms. These discrepancies are either false negatives (the algorithm failed to find a path) or false positives (the algorithm inferred a path that wasn't there).



Summed edge-counts represent the combination of paths for accidentally duplicated message deliveries due to a mail server configuration error at node 38.

Figure 16: Received-header trace results (convolution algorithm)

We can compute false positive or negative ratios based on counts of path patterns, path instances, or messages. For example, if the actual system executed the path $A \rightarrow B \rightarrow C \rightarrow D$ twice, but the algorithm found one instance of $A \rightarrow B \rightarrow C \rightarrow D$, one instance of $A \rightarrow B$, and one instance of $C \rightarrow D$, then:

- Counting path patterns, the algorithm had no false negatives and two false positives.
- Counting path instances, the algorithm had one false negative and two false positives.
- Counting messages, the algorithm failed to ascribe 3 of 6 messages to the correct path (false positives = false negatives, in this case).

Of course, to compute these ratios we need a representation of the ground truth. Fortunately, the nesting algorithm is able to produce guaranteed correct paths if we "cheat" and tag each trace message with a path-instance-ID value. We can do this for our synthetic traces (from Maketrace) and our PetStore traces, so we can run the algorithm once with path-IDs and once without, and compare the results. We can also extract exact paths from the Received-header trace, using Message-ID email headers, and compare those paths to the output of the convolution algorithm. We cannot, unfortunately, evaluate false positive or negative ratios for other kinds of traces (e.g., those obtained by packet sniffing).

Our algorithms tend to fail by generating a large variety of false

positive path patterns with low instance counts, among a set of accurate patterns with high counts. Recall that our primary goal is to identify the most frequently executed paths in the distributed system, so a useful tool will rank-order the inferred path patterns by count and then prune away most of the low-frequency path patterns. Therefore, we examine whether the top N patterns that remain after pruning match the top N ground-truth patterns, or whether pruning causes some of those top N ground-truth patterns to be omitted. For a given algorithm and trace, we can plot the number of omitted ground-truth patterns as a function of N.

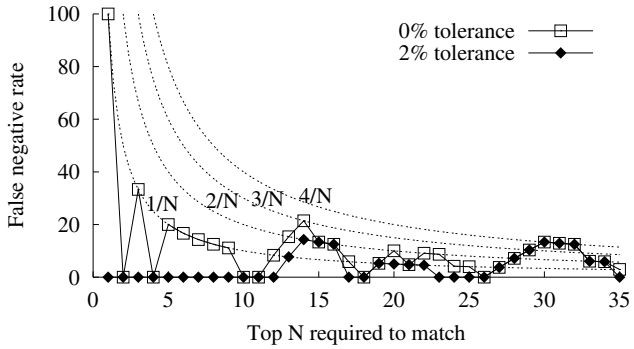


Figure 17: False negative path pattern rate vs. pattern pruning

Figure 17 plots the results for the nesting algorithm over a maketrace-generated trace for the multi-tier configuration. (This trace included 202,498 messages with a mean parallelism of 42.) The false-negative rate in the figure is bounded in most cases by $1/N$, indicating that most false negatives are the result of near-ties in the ranking. (Our belief that these are really near-ties is confirmed by the points plotted for "2% tolerance," where we ignored false negatives whose instance count was within 2% of making the top N. At 6% tolerance, almost all false negatives disappear.) Several true false positives (at ranks 27 and 28) cause additional false negatives for high values of N, by displacing true positives. We ran the same tests for simpler synthetic traces and found no false negatives except in the case of a few near ties.

We ran similar experiments for the convolution algorithm. It also found the top N paths, with similar errors in the case of ties.

Our goals also include accurate measurement of path-specific latencies internal to nodes. Accurate path inferences are obviously a prerequisite, so we have placed more emphasis on quantifying path inference accuracy. However, we have found that even with relatively high error rates in inferring paths, the latencies we find for correctly-inferred paths are within a few percent of the correct values.

7.5.2 *Testing using pathological cases*

Certain special combinations of causal paths can cause our algorithms to make false inferences, especially when many paths are being executed in parallel. We devised a number of pathological cases, depicted in Figure 18, on which we could test the accuracy of our algorithms:

Children-parallel has B calling C twice in parallel. This breaks all three of the nesting algorithm's child-penalty heuristics. In most situations, it is our worst case, but it becomes one of our best cases when delay deviations are low, or when call parallelism is high.

Children-0/2 has node B calling node C twice in series in one pattern, while the other pattern has no calls to C. This was a hard case for a simpler version of the nesting algorithm that lacked a scoreboard, and so could not assign both C calls to the same pattern.

Children-d/cc has node B calling node C twice in series in one pattern, and B calling D once in the other. This is a hard case for the nesting algorithm, especially with high parallelism, because the child-penalty heuristics wrongly lead the algorithm to assume that B is calling C and D in series.

Penalty-breaker includes two paths with multiple calls to the same child, and one with no such call. Also, the delays on the two longer paths are identical, causing lots of confused assignments. This breaks two of three child-penalty heuristics, and inspired the third one. It demonstrates the tradeoffs required when selecting default values for the three penalties.

We use these test cases, as well as synthetic multi-tier traces, in the next few experiments evaluating the accuracy of the nesting algorithm.

7.5.3 *Testing the effects of parallelism*

As parallel activity increases in a trace, the nesting algorithm has a harder time correctly assigning calls to paths. The result is an increase in the number of false-positive path instances inferred (which can push true paths out of the top N). We ran a series of experiments with increasing parallelism to see how this affects the false-positive rate; Figure 19 shows the results. The X axis in this figure is a dependent variable, roughly linear with the average amount of parallelism used by the trace generator.

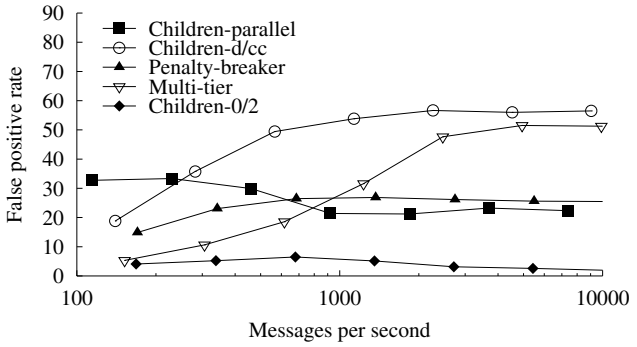


Figure 19: Effect of trace parallelism on nesting algorithm accuracy

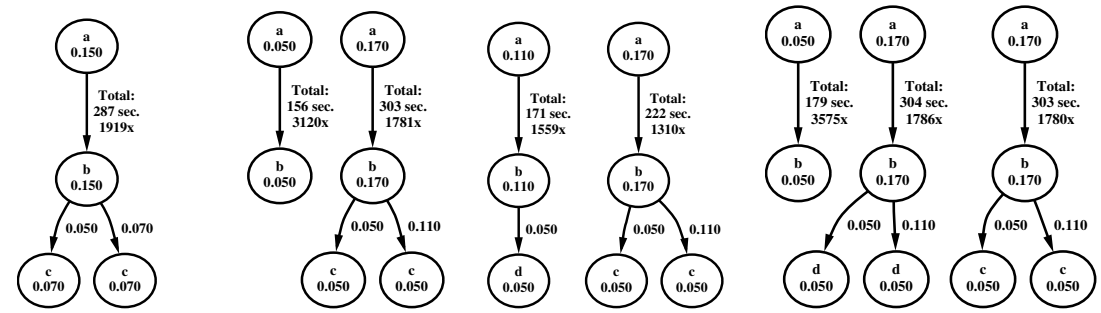
Generally the algorithm's performance is moderately worsened by increasing parallelism. The effect on the multi-tier case is somewhat more pronounced. For the Children-parallel case, increasing parallelism actually improves performance, perhaps because the algorithm has more opportunity to infer that the child calls are in parallel.

7.5.4 *Testing the effects of delay variation*

Maketrace generates random delays at each node in a call, using a Gaussian distribution. We can vary the standard deviation of these distributions to see how increasing delay variation affects the false-positive rate for path instances; Figure 20 shows the results for the nesting algorithm. Generally, performance worsens with increasing variation. The Children-parallel and Children-d/cc cases are especially vulnerable to variation. Note that in a real system, one would not expect all of the node delays to have the same variance.

7.5.5 *Testing the effects of message loss*

We expect our tools to be used with traces collected by passive network sniffing, which is often lossy. We can quantify the effects of message loss on the accuracy on our algorithms, by comparing



(a) Children-parallel (b) Children-0/2 (c) Children-d/cc (d) Penalty-breaker

Figure 18: Causal path pattern combinations for pathological cases

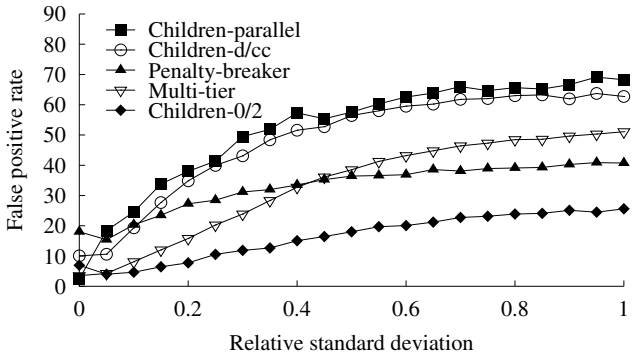


Figure 20: Effect of delay variation on nesting algorithm accuracy

the results of an algorithm on a loss-free trace and on a similar trace with randomly deleted messages.

The maketrace generator allows us to model the bursty losses typical of network sniffing. The program models a sniffer with a peak capture rate and a finite queue, and discards packets that would overflow this queue.

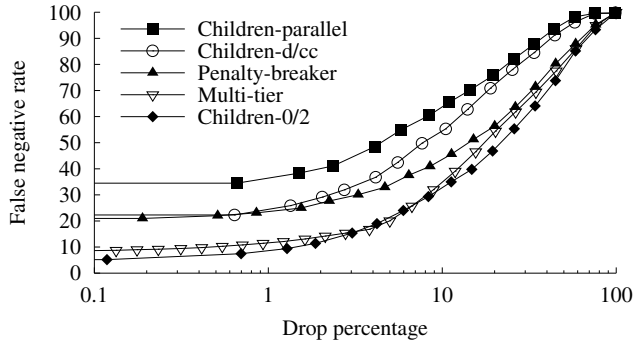


Figure 21: Effect of message drop rate on nesting algorithm accuracy

We tested the effect of message drops on the performance of the nesting algorithm, using several synthetic traces. Figure 21 shows the results of numerous trials. Performance is expressed as the fraction of false negative path instances. We fixed the maximum queue length at 64 packets, and increased the peak capture rate in each trial until the drop rate reached zero. The necessary capture rates ranged from 115 to 605 packets/sec. (Note that the message rates in our simulations are arbitrary and relatively low.) Each point in the figure corresponds to a specific capture rate; thus, both the false-negative rate and the drop rate are dependent variables.

The results show that for low drop rates (below about 1%), algorithm performance is unaffected. For higher rates, but below about 10%, performance is reduced but not unacceptable. At higher drop rates, it is not surprising that the results are bad. A real tracing system, therefore, must be sufficient to capture most packets, but need not be perfect.

7.5.6 *Testing the effects of clock skew*

To test the effects of clock skew on traces collected at more than one point in the network, we wrote a simple program, "skewer," to add per-node clock skew to an existing trace. We can then perturb an unskewed trace by varying amounts to test how our tools cope with skew. (Skewer is also useful to de-skew a real-world trace; we use additional information, such as that obtained from NTP, to remove the mean per-node skews from a multi-point trace.)

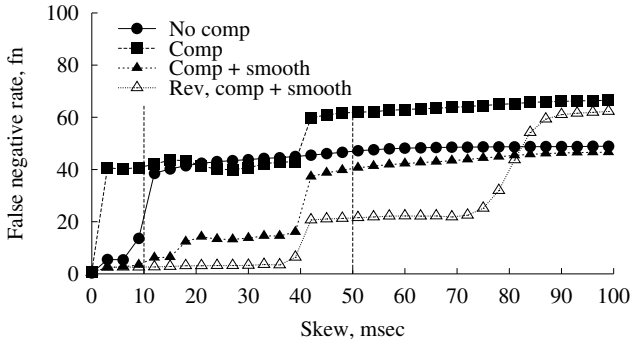


Figure 22: Effect of clock skew on nesting algorithm accuracy

The nesting algorithm includes two features for compensating for skew. First, it supports a configurable skew window tolerance, which loosens comparisons between timestamps where they are used to establish nesting relationships. Second, it allows smoothing of each scoreboard entry, which widens peaks to make the parent-selection step more tolerant. Figure 22 shows the multi-tier trace with varying skew added at the *WS2* node (i.e., *WS2*'s clock runs 0 to 60 msec. fast), and a fix skew window of 30 msec. The "no comp" curve shows inaccuracy (false-negative rate) without skew compensation. The "comp" curve shows inaccuracy with only the skew window enabled; this actually decreases accuracy. The "comp+smooth" curve shows the combination of skew windows and smoothing, which performs best for most reasonable levels of skew. The "rev, comp+smooth" curve shows what happens if *WS2*'s clock instead runs slow, rather than fast, by the value on the x-axis. The vertical marks indicate 50 msec. and 10 msec.; these are the mean call and return delays, respectively, in the trace. The nest-

ing algorithm results in few false negatives when skews are smaller than the sum of the skew window (i.e., one's estimate of worst-case skew) and the actual delays, but performs badly for larger skews.

7.5.7 *Accuracy of the convolution algorithm*

We ran trials of the convolution algorithm on the Received-header trace, varying the time quantum (μ) from 5 secs. to 720 secs. We compared the output to a ground-truth graph extracted directly from the email messages. Over all of the time quanta we tried, the false-positive rate varied between 21% and 29%, with only minor dependence on the quantum (setting $\mu \geq 360$ yielded the worst results). However, if we ignore paths that are reported with fewer than 100 messages, the false-positive rate drops to zero, except for $\mu = 720$. (Note that such large μ values are useless, in any case, for finding non-zero delays in this trace.) In no case did the algorithm miss any frequent real paths in this trace (i.e., the false-negative rate for frequent paths is zero).

7.6 Results: Execution costs

We measured run time and memory costs for the experiments in the previous sections. Note that neither program has been fully optimized, and the convolution algorithm presents several tradeoffs between accuracy and speed that may require some trial and error.

Table 1 shows the costs. **Length** gives the trace length in messages; **Duration** gives the elapsed time of the trace; **MBytes** gives the amount of data space allocated (not counting stack or code); **CPU secs.** gives the user-mode CPU time (kernel mode is negligible in all cases). The table also shows the (computed) **mean per-node parallelism** for the nesting algorithm, and the time quantum (μ) for the convolution algorithm. We ran the nesting algorithm on a 1.7 GHz Pentium 4 running Linux 2.4.20, and the convolution algorithm on a 667 MHz AlphaServer running Tru64 UNIX V5.1.

We ran experiments to verify the scaling properties described in Sections 5.1.2 and 5.2.5. The nesting algorithm's run-time and space requirements should be $O(mp)$, where m is the trace length in messages and p is the mean per-node parallelism. Table 1 includes rows for several different trace lengths for each of two systems, and several p values for one system, to illustrate these effects. Costs are not quite linear in p, probably due to certain constant space overheads for small p, and poor locality for large p.

The convolution algorithm's run-time is mostly dependent on the trace duration and time quantum, and not much on the trace length. Figure 23 shows CPU time measurements for the Received-header trace, at various time quanta. The figure shows that these measurements fit the $S \log S$ curve, where S is the trace duration T divided by the time quantum. We did not run the convolution algorithm on the longest traces in Table 1; with our current resources, the algorithm's run-time becomes prohibitive if the trace duration is more than about 100,000 times the desired time precision (i.e., the time quantum).

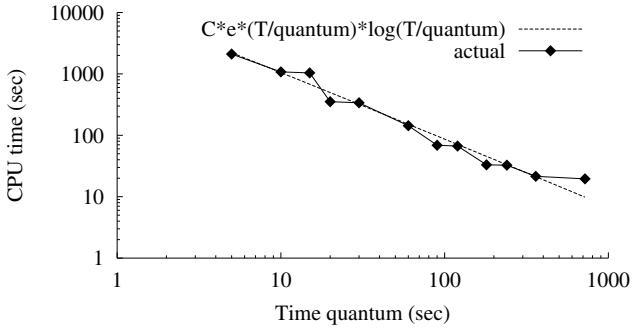


Figure 23: Convolution run time vs. time quantum (Received-header trace)

8. FUTURE WORK

The most important remaining work is to trace and analyze full-scale, real-world applications with our tools. We are negotiating with owners of several such applications for access to their systems, but privacy issues and concerns for both proprietary data and system stability have slowed progress. We expect experiments with these traces will force us to improve the competence and efficiency of our algorithms, and to automate or settle the choice of free parameters. Each new trace we have received so far has led to improvements in our algorithms.

We are extending our tools to add several significant capabilities, including techniques for locating the causes of low-frequency high-latency end-to-end behaviors. We would also like to extend our techniques to handle lock-based interactions between nodes; we want to know not only that node A on call path $P1$ often waits for lock L, but also that L is usually being held in these cases by node B on path $P2$.

We plan to develop a sliding window version of the nesting algorithm that processes all messages within a time window into path pattern instances, before processing messages in the next (overlapping) time window. This modification would solve two problems: (1) If the system demonstrates phased behavior, where some pattern is frequent only in a short time interval but infrequent over the whole trace, that pattern may be hard for the user to notice among all of the other low-frequency patterns. However, if the pattern is relatively frequent in one time window, then it could be much easier to spot. (2) The algorithm scales linearly in memory usage with the number of messages, but it cannot currently handle traces with greater than a few million messages and a lot of parallelism. Windowing would allow processing of much more complex traces.

The nesting algorithm produces a set of distinct causal paths. One might want to merge similar paths to form a single visualization of the system as a graph, where an edge between two nodes shows the probability of a corresponding call pair. Carrasco and Oncina [4] describe an algorithm that might work to merge paths.

So far, we have only partially addressed the visualization problem, but any truly useful tool will require clever rendering of the outputs of our algorithms. Both Magpie [13] and NetLogger [26] provide simple visualizations, but it is not clear if these are right for our purposes. The Critical Path Analysis technique of Yang and Miller [19] might also prove useful when applied to the outputs of our algorithms.

9. SUMMARY

We proposed an approach to performance debugging for distributed systems. It differs from prior approaches by adopting as strict a "black-box" model as possible, and through the use of low-level traces, little semantic knowledge, passive monitoring, and offline processing. We have developed two distinctly different algorithms, each with their own strengths and weaknesses. Preliminary results, based on several different kinds of traces, suggest that the tools do produce useful and accurate results, and we are now working on testing them with more real traces.

10. ACKNOWLEDGEMENTS

We wish to thank Steve Langdon for inspiring this research; Hank Jakiela, Whit Turner, and Richard LaPerle for their assistance in trace collection; Emre Kiciman for providing and helping

Trace	Length (messages)	Duration (secs.)	Nesting algorithm: Mean per-node parallelism	Nesting algorithm: MBytes	Nesting algorithm: CPU secs.	Convolution algorithm: μ (secs.)	Convolution algorithm: MBytes	Convolution algorithm: CPU secs.
Multi-tier (short)	20,164	50	1.793	1.5	0.23			
Multi-tier "normal"	202,520	500	1.641	13.8	2.27	0.01	0.2	6684
Multi-tier "added-delay"	196,438	500	1.744	13.4	2.31	0.01	0.2	6709
Multi-tier (long)	2,026,658	5000	1.612	136.8	23.97			
Multi-tier, parallelism-low	769,638	5,000	1.146	54.0	7.54			
Multi-tier, parallelism-medium	770,344	500	5.116	54.2	11.15			
Multi-tier, parallelism-high	775,254	50	45.057	132.1	233.61			
PetStore "normal"	252,024	1,999	1.322	19.8	3.34	0.02	26	12780
PetStore "const-delay"	234,036	2,000	1.313	18.4	2.92	0.02	25	6301
PetStore "normal" (full)	1,345,538	10,799	1.331	97.1	17.12			
PetStore "const-delay" (full)	1,288,223	10,799	1.318	93.2	16.41			
Email headers	81,044	5.1×10^6				5	131	2106
Email headers	81,044	5.1×10^6				30	36	338

Table 1: Execution costs

us with his J2EE tracing system; Jun Li for providing more traces; John MacCormick for good ideas; Dawson Engler for shepherding; and the anonymous reviewers for their probing comments.

11. REFERENCES

[1] Alignment Software, Inc. Appassure. http://www.alignmentsoftware.com, 2003.

[2] S. Bagchi, G. Kar, and J. L. Hellerstein. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In *Proc. 12th Intl. Workshop on Distributed Systems: Operations & Management*, Nancy, France, Oct. 2001.

[3] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proc. 7th IFIP/IEEE Intl. Symp. on Integrated Network Management*, Seattle, WA, May 2001.

[4] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proc. 2nd Intl. Colloq. on Grammatical Inference*, pages 139–150, Alicante, Spain, Sep. 1994.

[5] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer. Using runtime paths for macro analysis. In *Proc. HotOS-IX*, Kauai, HI, May 2003.

[6] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic systems. In *Proc. 2002 Intl. Conf. on Dependable Systems and Networks*, pages 595–604, Washington, DC, June 2002.

[7] A. Feldmann. BLT: Bi-layer tracing of HTTP and TCP/IP. In *Proc. WWW9*, pages 321–335, Amsterdam, May 2000.

[8] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, Sept 1999.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Proc. SIGPLAN Symp. on Compiler Construction*, pages 120–126, Boston, MA, June 1982.

[10] J. L. Hellerstein, M. Maccabee, W. N. Mills, and J. J. Turek. ETE: A customizable approach to measuring end-to-end response times and their components in distributed systems. In *Proc. ICDCS*, pages 152–162, Austin, TX, May 1999.

[11] C. Hrischuk, J. Rolia, and C. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In *Proc. MASCOTS '95*, pages 399–409, Durham, NC, Jan. 1995.

[12] P. Huang, A. Feldmann, and W. Willinger. A non-intrusive, wavelet-based approach to detecting network performance problems. In *Proc. Internet Measurement Workshop*, San Francisco, CA, Nov. 2001.

[13] R. Isaacs and P. Barham. Performance analysis in loosely-coupled distributed systems. In *7th CaberNet Radicals Workshop*, Bertinoro, Italy, Oct. 2002.

[14] V. Jacobson, C. Leres, and S. McCanne. `tcpdump`. www.tcpdump.org, 1989.

[15] JBoss Group. http://www.jboss.org/.

[16] E. Kiciman. JBoss request-tracing in Pinpoint, 2003.

[17] J. B. Micheel. Personal communication, 2003.

[18] B. P. Miller. Dpm: A measurement system for distributed programs. *IEEE Trans. on Computers*, 37(2):243–248, Feb 1988.

[19] B. P. Miller and C.-Q. Yang. Critical path analysis for the execution of parallel and distributed programs. In *Proc. 8th Intl. Conf. on Distributed Computing Systems*, pages 366–373, San Jose, CA, June 1988.

[20] D. L. Mills. The network computer as precision timekeeper. In *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, pages 96–108, Reston, VA, Dec. 1996.

[21] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proc. SIGCOMM '97*, pages 167–179, Cannes, France, Sep. 1997.

[22] Performant, Inc. Optibench. http://www.performant.com/.

[23] Quest Software Inc. Performasure. http://java.quest.com/performasure, 2003.

[24] Sun Microsystems, Inc. Java Pet Store Demo. http://developer.java.sun.com/developer/releases/petstore/.

[25] Sun Microsystems, Inc. J2EE platform specification. http://java.sun.com/j2ee/, 2003.

[26] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. IEEE High Performance Distributed Computing Conf. (HPDC-7)*, July 1998.

[27] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proc. 9th USENIX Security Symp.*, Denver, CO, Aug. 2000.