Towards Automatic Inference of Task Hierarchies in Complex Systems

Haohui Mai°, Chongnan Gao†, Xuezheng Liu‡, Xi Wang§, Geoffrey M. Voelker *
‡ Microsoft Research Asia, † Tsinghua University, * University of California, San Diego
° University of Illinois at Urbana-Champaign, § MIT CSAIL

1 INTRODUCTION

As Web services increasingly weave their way into our everyday lives, their dependability has become of critical importance [16]. Software defects, however, continue to plague such services, contributing to either degraded performance or down time. The most difficult bugs are the ones that make the system deviate from its expected execution behavior. Their root causes are usually buried in convoluted application logic, making their detection and analysis difficult.

The inherent complexity of such systems further obstructs understanding such unexpected runtime behavior. Typical Web services involve multiple tiers, a hierarchy of functional abstractions, and high degrees of concurrency. They usually execute user-level *tasks*, e.g., processing client requests, in a series of stages, which can be distributed across multiple machines, processes, and threads, using events and asynchronous messages as notification mechanisms. Verifying the behavior of such individual tasks therefore becomes a very challenging problem since developers have to reconstruct the task flow by linking together pieces of its execution throughout the system.

Conceptually, developers can represent such execution as a *hierarchical task model* that is consistent with the layered design of the system. Each task in the hierarchy represents the execution of a component or a stage at a particular layer. A task at a higher layer executes by completing several dependent *child tasks* at lower layers. For instance, Figure 1 illustrates the task hierarchy of committing user data in PacificA, a distributed storage system [12] similar to Google BigTable [6]. The top task has two sequential child tasks corresponding to the local and the remote commit. Each child task issues two parallel RPC calls as lower tasks to storage nodes, and an RPC call further invokes `send` and `recv` functions as tasks inside the network library for passing messages. At the bottom layer, each *leaf task* corresponds to sequential execution of code between *synchronization points*. Based on the task model, developers can better understand the structures of components and their dependencies, and use debugging tools (e.g., Pip [15] and D^{3}S [13]) to instrument the system and verify the behavior of tasks at appropriate layers.

However, existing tools require developers to *manually* specify task models. For example, Pip relies on developer-provided "expectations" (a hierarchical task model with assertions on resource usage) to validate system behavior. Writing a comprehensive task model that covers both high-level design and enough low-level implementation details can be tedious and error-prone, especially for a large system that may evolve rapidly. Path-based tools such as Magpie [3] can infer per-request causal paths from event traces, but are restricted to tracking predefined events and rely on developers to specify task boundaries in the event schema.

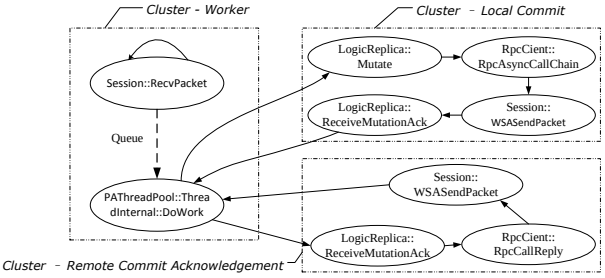

Figure 1: A hierarchical task model of committing user data in PacificA. The nodes are leaf tasks labeled by function names, with edges showing causal dependencies among them.

The goal of this position paper is to explore the extent to which hierarchical task models can be automatically inferred, with minimal or no manual assistance from developers. Developers would no longer have to manually annotate source code or specify event boundaries, a necessary step towards fully automating the diagnosis of complex systems. Automatic inference of task models from a running system can also provide more complete coverage. As a result, developers and system administrators can use these task models to visualize the design and implementation of a system, and use the models as input into other debugging and verification tools.

An automatic inference tool for task models faces several challenges. First, it must identify appropriate task boundaries based solely on monitored execution behavior, without requiring explicit annotation from developers. Second, it must correctly associate dependencies among tasks. In particular, the tool must be able to identify dependencies of tasks that share resources (e.g., thread pools, locks) and identify useful causalities. Finally, it must recover the hierarchical structure among tasks — a task may call several child tasks, either sequentially or in parallel. Given the non-deterministic nature of task executions in complex systems, determining their hierarchical structure is non-trivial.

In this paper, we describe a framework for automatically inferring hierarchical task models in complex sys-

tems, a debugging tool called Scalpel that implements this framework, and the use of Scalpel on two systems as a feasibility case study. In our framework, we transparently instrument a running service to collect execution traces of function calls and system calls of synchronization points, such as signal/wait on events. We use these traces as input into an inference engine for creating a hierarchical task model of the service. The inference engine uses a bottom-up approach across three stages, each addressing one of the above-mentioned challenges.

First, we partition the execution traces into separate leaf tasks whose boundaries correspond to these synchronization points. Synchronization points identify where causal dependencies among threads and processes occur, and therefore are good heuristics for identifying task boundaries.

Second, we connect leaf tasks together into a directed *causal graph*, where an edge between two task nodes signifies that the execution of one task casually depended on a previous task (e.g., sending a response message in reaction to a request). We infer causal dependency using the happens-before relation among tasks.

Finally, we overlay a hierarchical structure on the task graph, where each layer in the hierarchy roughly corresponds to a design or implementation layer of the system. We use clustering to detect repeated patterns (i.e., subgraphs) in the causal graph to identify executions of child tasks (e.g., sending messages) of higher-layer tasks (e.g., sending an RPC) and recursively apply the algorithm to identify successively higher layers.

Our preliminary experience with Scalpel is encouraging. We have applied it to two systems, the Apache Web server and the PacificA distributed storage system, and have found that Scalpel can automatically infer meaningful task models without any annotation or schema provided by developers. Furthermore, Scalpel enabled one developer to track down a performance bug in PacificA that degraded network throughput by up to 30%.

The rest of the paper is organized as follows. Section 2 presents the task inference framework, and Section 3 describes the use of Scalpel on Apache and PacificA as a feasibility case study. Section 4 discusses related issues. We survey related work in Section 5 and conclude in Section 6.

2 DESIGN

In this section, we describe the design of our framework as implemented by Scalpel.

2.1 Collecting Traces

Scalpel collects traces of function calls and their parameters, including synchronization calls (e.g., `signal` and `wait`) and OS socket calls (e.g., `send` and `recv`). By default all functions are traced, although in practice we can use heuristics to prune functions that have no impact on inferring the hierarchy. Tracing is transparent to the system and relatively lightweight.

Typically systems have a custom logging facility which can contain additional useful information for determining task boundaries. This information could be manually incorporated, e.g., as done by Magpie for creating log schemas. We defer this to future work.

2.2 Identifying Leaf Tasks

With the execution traces as input, the baseline step is to identify the boundaries of leaf tasks. Scalpel uses *synchronization points* as heuristics for defining the boundaries. A synchronization point is where two threads synchronize their execution and establish a happens-before relation: it can be either for mutual exclusion (e.g., a thread releases a lock before another thread acquires the lock) or for coordination (e.g., a thread signals an event to a waiting thread, or sends a message to another thread on a remote machine).

We refer to the execution between two consecutive synchronization points as a "continuation". We consider the work performed at the granularity of a continuation as relatively independent and self-contained. As such, we consider continuations to be a reasonable representation of the smallest unit of work in a task model and are a natural definition for a leaf task. Further, since synchronization points define the boundaries of leaf tasks, dependencies among them only occur at their boundaries.

Scalpel currently identifies synchronization points by instrumenting the appropriate library and system calls for synchronization primitives (locks, events, etc.) and sockets (we consider communication as synchronization). For the systems we have experimented with, this instrumentation has been sufficient. If a system uses spin-locks or lock-free data structures, however, synchronization library and system call instrumentation is not sufficient. As a result, Scalpel will define larger leaf tasks that span these user-level synchronization points — whether the larger granularity fundamentally degrades the utility of the task model for debugging remains an open question. Manual identification of user-level synchronization primitives would solve the problem, but would no longer be automatic. We speculate that heuristics looking at call-graph patterns and types could help, but we have not experimented with any approaches and so it simply remains speculation.

2.3 Constructing Causal Graph

To coalesce tasks into higher layers, we first need a representation of the relationships among all the leaf tasks. Scalpel uses a directed causal graph to reflect leaf task relationships. In this graph each edge represents a causal

dependency between two dependent leaf tasks, which execute in succession to accomplish a logically higher-layer task. For example, in PacificA when a primary machine receives a commit message, it first stores the data locally (`LogicReplica::Mutate`) and then hands over the data to the secondary node via asynchronous RPC (`RpcClient::RpcAsyncCallChain`). Therefore, an edge is created to connect the two leaf tasks.

Scalpel infers causal dependencies using the happens-before relation, which includes executions of two successive leaf tasks by the same thread as well as executions of leaf tasks across threads at synchronization points. It is critical to distinguish true causal dependency from occasional "run-after" relations. For example, a task that consumes an event in a queue causally depends on the producer task that enqueues the event. On the other hand, when accessing shared resources exclusively threads need to be synchronized via mutual exclusion, which simply creates a run-after relation. We do not regard the mutual exclusion as a causal dependency because the order of thread executions is arbitrarily decided by the thread scheduler.

Scalpel uses several heuristics to identify true causal dependencies. When the system uses OS-provided queues (e.g., I/O completion ports, as used in Apache and PacificA) and notification mechanisms (e.g., events), Scalpel can construct the dependencies by matching the tokens between producer tasks and consumer tasks (the tokens are recorded in the traces as parameters to instrumented system calls). As OS mutexes and semaphores are used primarily for mutual exclusion, rather than thread coordination, currently Scalpel simply regards them as a run-after relation.

For socket communication with TCP, the byte stream abstraction blurs the message boundaries between sender and receiver. Currently, developers need to provide additional matching information (e.g., annotating the message handlers), otherwise Scalpel cannot precisely match the sender and the receiver of the same message, and may generate false dependencies between tasks. By adopting ideas from automatic protocol reverse engineering [4, 8], it may be possible to further eliminate this annotation requirement. However, we have yet to explore these possibilities.

2.4 Inferring Hierarchical Structure

To infer a hierarchical task structure from the causal graph of leaf tasks, we mine repeated patterns that occur in the causal graph. Our algorithm is inspired by whole program paths [11], which searches for "hot subpaths" (i.e., frequent subsequences in call paths) to infer a context-free grammar. Similarly, our algorithm searches for frequently occurring subgraphs. We consider each subgraph as logically representing a higher-layer task that is composed of the child tasks in the subgraph. Applying the algorithm recursively, we obtain a hierarchical task model.

The algorithm first enumerates all connected subgraphs (with more than one node) from the causal graph, and then clusters these subgraphs into different patterns based on their similarity (described further below). The algorithm considers each pattern that has a high number of occurrences as representing a higher-layer task. It then substitutes each subgraph having that pattern with a "super" node. To identify successively higher layers, Scalpel applies this algorithm recursively until no high-frequency patterns remain. At this point, the graph has several super nodes and possibly some unclassified leaf tasks, and could be connected or disconnected. Scalpel takes the set of non-trivial super nodes as the final result, each one representing an inferred highest-layer task model. By expanding the super-nodes into the subgraphs that formed them, we obtain the full task hierarchy.

The algorithm requires a similarity measure between subgraphs for clustering them into patterns. Currently we use exact matching between subgraphs, i.e., all subgraphs in the same cluster are isomorphic. We use a deterministic serialization method to encode a subgraph, and cluster the subgraphs by comparing the hash values of their encodings. This approach makes the clustering algorithm very efficient. When encoding leaf tasks, we use the function names on the call stacks at the beginning and the end of the task, and ignore other function calls in the middle of the task. For leaf tasks to fall into the same cluster, they must have identical call stacks of function names. This heuristic ignores unimportant differences among leaf tasks of the same kind, e.g., the different parameter values in functions and the thread context. In our experience, this approach works very well for identifying the same kind of leaf tasks.

2.5 Implementation

We have implemented Scalpel on the Windows platform using our library-based record&replay tool named R2 [10]. R2 instruments OS system calls and user functions to record their execution. R2 also tracks system calls on synchronization objects (e.g., mutexes and events) and sockets, and constructs the happens-before relations among threads and processes. With this information provided by R2, Scalpel can easily construct the call stacks of leaf tasks, and perform the analysis as presented in Section 2.

3 Case Study

In this section we preliminarily evaluate the effectiveness of Scalpel. It is not immediately clear how such a task inference tool should be evaluated, though. Ideally,

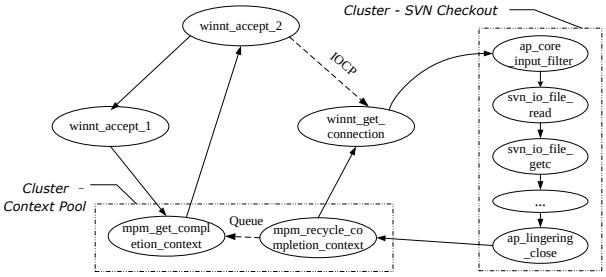

Figure 2: Hierarchal task model of Apache doing SVN checkout.

an effective tool will precisely capture developers' intuition, which can be difficult to quantify. Therefore, one approach is to simply let developers verify the inferred task models and determine whether they match their understanding of their systems. On the other hand, a task model is also a means to verify the correctness or performance properties of the system. Therefore, a second approach is to evaluate the effectiveness of the task models for debugging.

We use both approaches to evaluate Scalpel. We experiment on the Apache and PacificA systems, each configured in typical scenarios. We ran a Subversion (SVN) service on Apache, and used a client to perform ten SVN checkouts. We configured PacificA on two machines, which form a replication group with one machine as a primary and the other as a secondary. We ran a small test benchmark which creates a table and commits 15 items of random user data. We used Scalpel to trace their executions and infer task models for each system. We perform all experiments on machines with 2.0 GHz Xeon dual-core CPUs, 4 GB memory, running Windows Server 2003 Service Pack 2, and interconnected via a 1 Gb switch.

We first evaluate Scalpel by manually inspecting the inferred models. Figure 2 shows the result for Apache. Each node is a leaf task, and the solid and dashed lines between tasks show their causal dependencies (solid for same-thread and dashed for cross-thread dependency). To make the model more readable, Scalpel labels every leaf task with the state of the call stack at the beginning of the leaf task. For conciseness, figures in this paper only show names of the functions that invoke synchronization operations at leaf task boundaries.

By investigating the source code of Apache and SVN, we confirm that this inferred model is exactly the Apache service cycle for SVN checkout operations in which Apache accepts an incoming connection and uses a worker thread from a thread pool to invoke SVN service modules. Scalpel infers 5 leaf tasks in the Apache core (the left part of Figure 2). Each task represents a major step for Apache to accept and serve a client connection. Specifically, the listening thread accepts the connection (`winnt_accept_1`), fetches a connection context resource from a pool (`mpm_get_completion_context`), then posts the context to a worker thread through an I/O completion port (`winnt_accept_2`), and finally goes back to wait for future connections. At the other side of the I/O completion port, a worker thread from a pool will get the context (`winnt_get_connection`), and call the SVN service module to answer the request (from `ap_core_input_filter` which reads the message from the connection, to `ap_lingering_close` which closes the connection). After that, the thread recycles the connection context to the pool (`mpm_recycle_completion_context`) and goes back to the waiting status. We omit details of SVN checkout, but the tasks similarly capture each meaningful step during the process. This verifies that using synchronization points is an effective heuristic to identify task boundaries, for at least this system.

In addition, Scalpel successfully identifies two meaningful high-layer tasks, as denoted by rectangles (we manually give names to them according to their functionality). The first one (context pool) groups the tasks for recycling and fetching connection contexts. They together implement a resource pool for reusing connection contexts. The other one (SVN checkout) contains the entire list of tasks for a SVN checkout operation. Because these tasks are frequently called with the same causal dependency structure, Scalpel can precisely identify them as high-layer tasks.

Figure 1 shows the inferred task model for PacificA, an example of a hierarchical task model. The developers of PacificA confirmed that this model also characterizes the overall task of committing user data at a primary node in a precise and meaningful way. There are three major high-layer tasks: a worker task, a local commit task and a remote commit acknowledgement task. In the worker task, a socket worker which waits for a client's request (`Session::RecvPacket`) repeatedly runs as a loop to receive messages from clients, and triggers a thread pool worker (`PAThreadPool-::ThreadInternal::DoWork`) to run the task via a queue. The local commit task first stores the data locally (`LogicReplica::Mutate`) and then hands over the data to the secondary node via asynchronous RPC (`RpcClient::RpcAsyncCallChain`) through the network layer (`Session::SendPacket`), and finalizes this local commit (`LogicReplica-::ReceiveMutationAck`) by issuing an acknowledgement locally. The remote commit acknowledgement task handles the secondary's acknowledgement (`LogicReplica::ReceiveMutationAck`) and replies to the client via RPC (`RpcClient::RpcCallReply`) through the network layer (`Session::SendPacket`).

It is worth noting that Scalpel not only identifies the three major steps for committing data as high-layer tasks,

	Apache	PacificA
Leaf Tasks	423952	10636
Events	0	47
Mutex	210472	4950
IOCP	23	16
Socket	527	77
run-after	193972	11304
Running Time	69.56s	61.14s

Table 1: Statistics of running inference algorithm, in terms of the number of leaf tasks, number of dependencies for different types, and total running time.

but also successfully separates local commit from remote commit acknowledgement as distinct tasks. It is not that Scalpel "understands" their semantics; it is because these two tasks are triggered by `DoWork` tasks separately. Therefore, our frequent pattern mining algorithm can separate each task correctly.

Table 1 summarizes the inference algorithm by showing statistics of inferred tasks and their relations after every stage.

Second, we evaluate Scalpel by using the inferred models in a debugging scenario. The developers of PacificA encouraged us to try Scalpel on PacificA because its performance under stress tests was not satisfactory. The developers suspected a performance bug, but could not isolate the cause even after several attempts at function-level performance profiling. We used the model in Figure 1 to profile PacificA's performance. Our profiling tool measures performance for each leaf task in terms of latency, bandwidth, and CPU cycles, and aggregates this data in a per-task manner for each layer.

By profiling the commit operation in a stress test, we soon found a performance problem: the committing task could not saturate network bandwidth, while at the same time the CPU usage remained low. To pinpoint the problem, we used a a top-down approach with the hierarchical task model: we started from highest-layer tasks down to leaf tasks to identify the tasks that consumed most of the running time. We found that, when sending packets at high frequency, the sender threads (typically configured as 4 threads) will block at a sleep function for 1 second (see source code in Figure 4) due to the flow control at the network layer when the internal message buffer is full. This flow control causes most sending threads to sleep synchronously. As shown in Figure 3, four threads make progress at roughly the same time.

A detailed investigation of the code reveals the root cause of the bug. When the RPC layer uses asynchronous communication, there is no RPC layer flow control. Every thread will send messages in a non-blocking fashion until the internal buffer at the network layer fills and the network layer blocks the thread (Figure 4). This causes the synchronization effect: no matter how many RPC sending threads are used, they will all be blocked by the

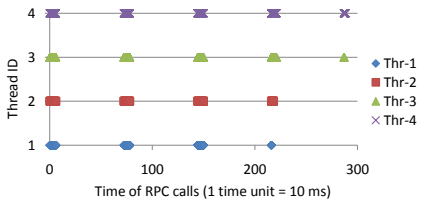

Figure 3: Trace of asynchronous PacificA RPC calls by 4 threads.

```
int Session::WSASendPacket(NetworkStream * pkt) {
    CAutoLock guard(_send_lock);
    while (_send_size > (64 << 20)) // 64 MB
        Sleep(1000);
    ...
    int rt = WSASend(_socket, buf, buf_num, &bytes,
        0, (OVERLAPPED*)ce, 0);
    ...
}
```

Figure 4: Flow control code in lower layers.

network layer synchronously at high workload, thereby limiting performance. This bug is caused by poor interactions across software layers. With a clear hierarchical model, our tool helped us soon identify the location of the bug and also quickly understand its root cause.

We recognize that our evaluations are more anecdotal than systematic. But these preliminary results provide encouraging evidence that our approach to automatically inferring hierarchical task models can be useful both for helping developers understand a complex system and for facilitating profiling tools to debug a system.

4 Discussion

Scalpel is primarily designed for developers to understand and debug the systems they build. Therefore, we assume that debugging information (e.g., symbols names) is available for Scalpel to trace user functions and provide readable names of leaf tasks. However, this requirement is not necessary for generating task structures. With only stripped executables, we can still instrument system calls to track synchronization points, and retrieve function addresses in corresponding call stacks via stack walking. These addresses can be used identically as function names in our analysis. Therefore, the inferred task models will have the same structure as before, only lacking names for leaf tasks. Without readable labels, these models may be difficult for use by developers but they could be sufficient for other debugging tools that can take advantage of a task-level representation.

Traditionally, people usually diagnose system behavior by means of causal path analysis [1, 3, 7], which models the execution as a collection of single-layer, causal paths. We believe that the hierarchical representation of tasks, adopted as a core part in Scalpel's design, is one step beyond causal path analysis. Indeed, at the leaf task layer our task model is the same as causal paths, and therefore can be used in similar path-based diagnoses

(e.g., optimizing end-to-end latency). However, the hierarchical representation can provide extra benefits in tackling system complexity. It encapsulates implementation details with high-level tasks, allowing developers to reason and verify system behavior at various task granularities. As a case in point, Pip has effectively detected bugs by checking user-provided properties on various levels of nested task structures. With our limited experience of Scalpel, we have not employed similar automatic property checking on the inferred task models. We will extend Scalpel in the future to support property checking, and further evaluate the benefits of the hierarchical task models compared with traditional causal path analysis.

5 Related Work

Path-based analysis. To analyze behaviors of multi-tier, distributed systems, previous work constructs causal paths from event traces generated at OS, network, and application levels. Magpie [3] takes user-provided schemas to correlate events into causal paths. Pinpoint [7] annotates applications to propagate a unique path identifier for each request. X-trace [9] extends network protocols with path-based metadata so as to maintain causal paths across network layers. These systems essentially require developers to provide application-specific task structures, in terms of task boundaries, correlation identifiers, and propagating rules. Scalpel automatically infers task structures through general heuristics, and does not require annotations.

Project 5 [1] and Sherlock [2] infer dependencies among network components from black-box network traffic. By doing so, they requires less effort than annotating source code. Scalpel also takes advantage of statistical inference, but further infers tasks hierarchies, and its tasks are not restricted to network components.

Data-flow tracking. Whodunit [5] and Data Flow Tomography [14] tracks data-flow among function calls by capturing all memory operations in an execution scope with virtual machines. While these systems provide precise causal dependencies, monitoring fine-grain memory operations can cause significant performance degradation. On the contrary, Scalpel only tracks synchronization operations and uses heuristics to infer causal dependencies. Therefore, it is more lightweight but provides less precise results. We see data-flow tracking and Scalpel as complementary approaches, and will investigate the possibility to combine them.

Predicate checking. Pip [15] and D^{3}S [13] check user-provided predicates to detect bugs. To define predicates, Pip requires structured behavior models which are close to the task models in Scalpel, and D^{3}S also needs to instrument specific functions to expose states for checking. Scalpel's task models can be used in these systems, making predicates easier to specify and check.

6 Conclusion and Future Work

Based on our initial experience with Scalpel on Apache and PacificA, we believe that hierarchical task models of complex systems can be inferred with few or no manual annotations. We plan to further investigate techniques for improving precision. We will extend our current trace collecting method, which currently ignores memory access operations (e.g., flagging and spinlocks), so that it is possible to perform a more comprehensive dependency analysis. In addition, we plan to explore more effective heuristics to prune occasional "run-after" cases to filter out unnecessary causalities. We are also interested in experimenting with various graph mining algorithms for recovering task hierarchies. Finally, we plan to apply Scalpel to a wide range of systems to more comprehensively evaluate our techniques.

References

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

[2] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*, 2007.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[4] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *CCS*, 2007.

[5] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *EuroSys*, 2007.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[8] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, 2007.

[9] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[10] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI*, 2008.

[11] J. R. Larus. Whole program paths. In *PLDI*, 1999.

[12] W. Lin, M. Yang, L. Zhang, and L. Zhou. Pacifica: Replication in log-based distributed storage systems. Technical Report TR-2008-25, Microsoft Research, 2008.

[13] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^{3}S: Debugging deployed distributed systems. In *NSDI*, 2008.

[14] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and visualizing full systems with data flow tomography. In *ASPLOS*, 2008.

[15] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[16] B. Stone. As Web Traffic Grows, Crashes Take Bigger Toll. `http://www.nytimes.com/2008/07/06/technology/06outage.html`, July 2008.