Lightweight Task Graph Inference for Distributed Applications

Bin Xin, Patrick Eugster, Xiangyu Zhang
Dept. of Computer Science
Purdue University
{*xinb,peugster,xyzhang*}*@cs.purdue.edu*

Jinlin Yang
Center for Software Excellence
Microsoft Corp.
jinliny@microsoft.com

***Abstract*—Recent paradigm shifts in distributed computing such as the advent of cloud computing pose new challenges to the analysis of distributed executions. One important new characteristic is that the management staff of computing platforms and the developers of applications are separated by corporate boundaries. The net result is that once applications go wrong, the most readily available debugging aids for developers are the visible output of the application and any log files collected during their execution. In this paper, we propose the concept of *task graphs* as a foundation to represent distributed executions, and present a low overhead algorithm to infer task graphs from event log files. Intuitively, a task represents an autonomous segment of computation inside a thread. Edges between tasks represent their interactions and preserve programmers' notion of data and control flows. Our technique leverages existing logging support where available or otherwise augments it with aspect-based instrumentation to collect events of a set of predefined types. We show how task graphs can improve the precision of anomaly detection in a request-oriented analysis of field software and help programmers understand the running of the Hadoop Distributed File System (HDFS).**

***Keywords*-task graphs; happens-before; distributed computing; log analysis; anomaly detection;**

I. Introduction

Large scale distributed applications running in third-party data centers have become increasingly popular due to the developments of search engines, e-commerce, and online social networks. Notable examples of such distributed services include Microsoft's Windows Azure, Google's App Engine and Amazon's Elastic Compute Cloud (EC2) platform [1].

The computing paradigm of large scale distributed applications presents new challenges to reliability. Because the administrative staffs of the platforms and the owners of applications running on them pertain to distinct corporate entities, developers of distributed applications are commonly limited to understanding execution and performing debugging of their code based solely on visible outputs and logs without the possibility of tapping into the execution. Traditional debugging practices such as stopping the application and attaching debuggers to nodes are thus hardly feasible.

Yet, a developer still needs to understand how his/her program proceeds when unexpected behavior occurs; how the control flows through different nodes; what the communication patterns and computing resource consumptions are when the application serves different kinds of requests; how a piece of high-level application logic ends up being executed in smaller pieces on different nodes.

The goal of the work described in this paper consists in developing lightweight techniques for forming a high-level structural view of distributed program executions to facilitate understanding and reasoning. Traditional approaches include representations of the structures and dependences of computation units in a program (static) or in an execution of the program (dynamic). For example, to analyze sequential programs, researchers have extensively used *control flow graphs* (CFGs) and static or dynamic *program dependence graphs* (PDGs). Research has been undertaken to extend these representations for parallel and distributed programs. For example, TraceBack [2] is a system that builds distributed control flow graphs with basic blocks and source line level details. However, because of CFGs' lack of inherent support for modeling *inter*actions between distributed processes as opposed to their fine-grained structure offered for *actions*, they are sub-optimal in distributed settings. Distributed executions can generate large amounts of CFG data. Without further abstraction, it is difficult for programmers to gain insights about the relevant portions of executions, and it is unclear whether such approaches can scale beyond simple scenarios such as web services implemented with a three-tier architecture.

Alternative models conversely center around threads-/processes and their interactions. Examples are vector clock-related approaches proposed as a means to preserve causality relations [3], [4], [5] between interleaving threads/processes. These approaches do capture interactions accurately but fail to link them to actions expressed by high-level programming constructs.

Our work aims at striking a balance between (a) conciseness of the data presented to programmers and (b) richness of relationships preserved between different pieces of computations through three contributions:

- We introduce the notion of *task* as abstraction for representing distributed executions. One can think of a task as an autonomous piece of computation that runs within a thread or process and has only limited and well-defined interactions with other tasks. These interactions, e.g., a signal on a semaphore or data received on a socket, induce task boundaries. *Task*

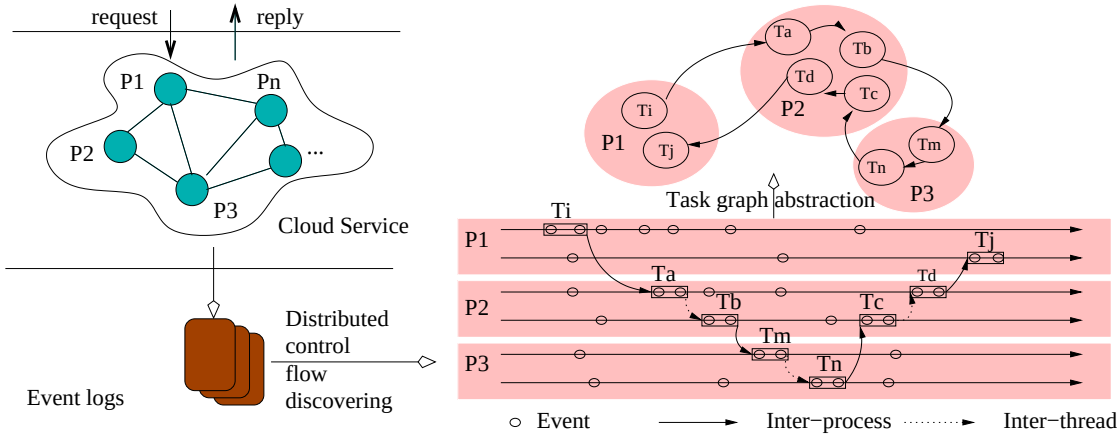


Figure 1. Schematic view of dynamic task graph inference for understanding distributed executions.

graphs are obtained by connecting tasks by pairing corresponding waits and signals or sends and receives.

- We propose a purely log-based light-weight approach to infer tasks and construct dynamic task graphs for executions of distributed applications.Event logging may already exist as an integral part of the application or can be augmented through aspects. We show how dynamic tasks are built by identifying two abstract types of primitive events: *acting* and *signaling* events.
- We develop a declarative formulation of task graphs by using Prolog. This formulation allows great flexibility in exploring inferred task graphs by writing simple Prolog queries. The output of such queries can then, for example, be visualized by running through tools like DOT, providing a valuable tool for program comprehension.
- We illustrate the benefits of inferred task graphs first by showing — through Hadoop, an open-source distributed file system implemented in Java — how they can aid in understanding and debugging distributed executions. Then we show how task graphs can improve anomaly detection [6] in distributed applications. We focus on *request-based analyses* of such applications, one of whose main goals is attributing resource consumptions (CPU, network) to request types, and show how task graphs can increase accuracy by reducing false positives (i.e. false anomaly alarms).

Note that the term “task” is quite overloaded and has appeared in a number of different research contexts, with diverging semantics. A good summary is given by Adve et al.[7]: 1. *tasks* are used as a programming construct to express parallelism;[1] 2. task *graphs* are used as an abstraction for performance modeling or 3. by compilers to partition programs and generate code for corresponding communication and by runtime systems for effective scheduling. In 1 and 2, tasks are known a priori and input by programmers. In 3, they are statically inferred by compilers. In this paper, task graphs are automatically inferred from event logs.

[1]Several more recently proposed programming languages designed for concurrent programming include abstractions of similar granularity as first-order entities (e.g. *activities* in X10 [8]).

II. Model and Overview

We assume the following model for the distributed systems that we are analyzing. A distributed application consists of a set of *processes* running on different nodes, connected pair-wise through reliable communication channels. Each process encompasses a number of *threads*. Threads communicate with each other through inter-thread synchronization mechanisms, e.g. semaphores, locks, or shared data structures, and across processes with inter-process communication mechanisms such as sockets or RPCs. Processes or threads can log events into files. This execution model accommodates realistic Java or C/C++ distributed applications.

Figure 1 gives a high-level system overview of what we propose. Users interact with a distributed application running in a computing cloud by sending requests and receiving replies. Event logs are collected independently on individual nodes and processed by our technique. The structure of the distributed execution is reconstructed by discovering the inter-process and -thread relations from the linear log files. Finally, a task graph is produced, describing the logged execution.

Figure 2 zooms in on the phase of inter-process and inter-thread relation discovery. The events in a log file (collected for a process) are first demultiplexed into per-thread event sequences; then, sequences of related events are abstracted into tasks (shadowed boxes); finally, interactions between events are established by matching event attributes, without demanding expensive vector clocks. The relations between tasks are abstracted from the relations between the events in the tasks. Since the second task in thread `T=35` starts thread `T=36`, there is an interaction from the second task of T=35 to the first block of T=36. Furthermore, thread `T=36`

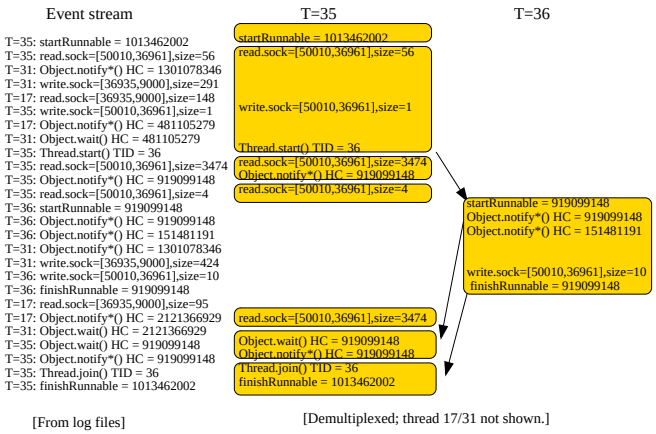


Figure 2. Zoom in on the "distributed control flow discovery" step in Figure 1.

signifies an object with id `919099148`, on which thread `T=35` is waiting, hence there is an interaction from `T=36` to the sixth task of `T=35`.

III. Tasks and Task Graphs

In this section, we present our definitions of events, tasks, and task graphs.

Definition 1: An *event* in a distributed program execution is the execution of an operation that sends (receives) a signal or data to (from) a different process/thread. There are two types of events: *acting events* and *signaling events*. A signaling event passes a signal or data to a different thread/process; the reception of this signal or data is called the acting event; it enables the receiving thread/process to proceed.

Events are the smallest building blocks of our system. Intuitively, a signaling event is the producer of a signal or data; an acting event is the consumer of a signal or data. For instance in Figure 2, in the per-thread log of `T=35`, at the end of the first shaded block, the event of "`Thread_start() TID=36`" is a signaling event and the corresponding acting event is the first log entry in the receiving thread (`T=36`).

Property 1: Every acting event must have a unique corresponding signaling event; a signaling event can have zero or multiple corresponding acting events.

The *happens-before* relation [3], $\longrightarrow_e$, is a partial order over the set E of events of an application execution such that for $a, b \in E$, $a \longrightarrow_e b \Leftrightarrow a$ causally precedes b. Relation $\longrightarrow_e$ thus defines the set of such tuples (a, b); we refer to these as *happens-before (instances)* (HBs). More specifically, in this paper relation$\longrightarrow_e$ is defined between an acting event AE and its corresponding signaling event SE, that is $SE \longrightarrow_e AE$.

In order to reconstruct system-wide task graphs, it is important to identify all inter-thread and inter-process events reflecting causality and data dependences. Table I shows the primitive events considered in Java-based systems. Shared variable reads and writes are not considered (directly) in the table. If such reads and writes occur in a way following specific synchronization then the synchronization pattern is captured by other means, such as the acquiring or relinquishing of a lock, leading to signaling and acting events.

Definition 2: A *task* is an autonomous computation inside a thread delimited by a pair of acting events, $[AE, AE')$, consisting of all the logged events between these two events inside the same thread.

Acting events inside a thread divide the whole computation of that thread into tasks. A task starts with an acting event and ends right before the next acting event. Conceptually, it represents the execution enabled by the signal/data received by the start event. Note that tasks are a dynamic concept. The following property can be directly inferred from the definition.

Property 2: A task must contain one acting event and zero to multiple signaling events.

The happens-before relation between *tasks*, $\longrightarrow_t$, can be defined based on the HBs between *events*, i.e., $t_1 \longrightarrow_t t_2$ if two events in the two respective tasks t_1 and t_2 comprise an HB. For instance, thread `T=36` in Figure 2 has one task as there is only one acting event in this thread. Denote the task as t_1^{36}, i.e., the first task in the thread. There are a number of HBs involving t_1^{36}: $t_2^{35} \longrightarrow_t t_1^{36}$, $t_1^{36} \longrightarrow_t t_6^{35}$ and $t_1^{36} \longrightarrow_t t_7^{35}$.

Definition 3: A *task graph* is a directed acyclic graph (DAG), whose nodes are tasks from all threads in the system, and edges represent HBs between these tasks.

Later on, we will conduct experiments of *request-based analyses* of distributed systems. We formally define the concepts of requests and replies as follows.

Definition 4: A *request* is a pair of signaling and acting events, with the signaling event originating from outside the system, while the acting event happens inside the system. The task starts with the acting event represented as T_{req}. A *reply* usually associated with a request is also a pair of signaling and acting events, but with the scope of the two events reversed, i.e., with the signaling event inside the system and the acting event outside. The task ending with the signaling event is represented as T_{rep}.

With the definitions of request and reply, we can now define *end-to-end* (E2E) request service graphs. The arrow $\overset{*}{\longrightarrow}_t$ represents the transitive task level HBrelation.

Definition 5: A E2E request service graph for a request req and corresponding reply rep is a task graph constructed from the set of tasks, $T = \{T_x | T_{req} \overset{*}{\longrightarrow}_t T_x \wedge T_x \overset{*}{\longrightarrow}_t T_{rep}\}$, and their HBs.

Our design choice of using acting events as task boundaries is essential to correctly attributing events to requests. Consider the example in Fig. 3, which shows two threads. `T1` receives two requests, r_1 and r_2, and then delegates them to `T2`. The request r_1 starts task Ⓐ in `T1`. The task involves all events until the next request r_2 is received. Task Ⓐ

Table I
SAMPLES OF PRIMITIVE EVENTS FOR JAVA BASED SYSTEMS. * I/O STREAMS CREATED OVER SOCKETS.

Categories	Acting	Signaling
Semaphore	`Object.wait` unblock,	`Object.notify[All]` call,
Threading	`Thread.run` start, `Thread.join` return,	`Thread.start` call, `Thread.run` finish,
Other (JCU)	`Condition.await` unblock, `Runnable.run` start, `Callable.call` start, `Future.get` return, `BlockingQueue.[take|poll|remove]` return	`Condition.signal[All]` call, `Executor.execute` call, `ExecutorService.submit` call, `BlockingQueue.[add|offer|put]` call
Inter-process	`SocketChannel.read`, `InputStream.read`*	`SocketChannel.write`, `InputStream.write`*

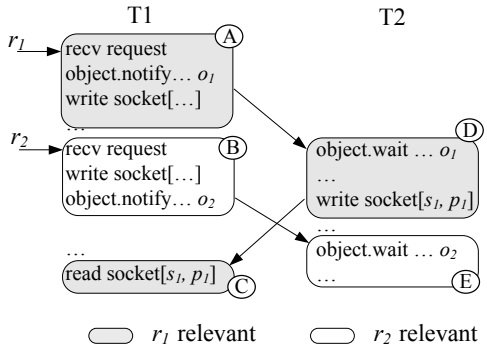


Figure 3. Intuition of task definition.

has a HB edge with Ⓓ due to the object notify/wait. Our technique does not introduce a HB edge between Ⓐ and Ⓑ like most vector-clock based approaches do. The intuition is that distributed systems implementations are mostly event driven, the semantics of execution originate from events from outside, namely, the acting events. Observe that later in `T1`, task Ⓒ is again r_1 relevant, due to the socket read/write. In comparison, if edges were introduced between tasks in the same thread, all tasks would become r_1 relevant.

IV. TASK GRAPH CONSTRUCTION

A. Acquiring Events and HBs

We leverage *aspect-oriented* techniques [9] to emit necessary logging data for task graph construction. Aspects allow for the type of interested events to be specified very generally, without specific programs in mind, and thus, permit reuse with different distributed programs. We use AspectJ to instrument Java-based distributed applications. The details are emitted for brevity.

Log files are parsed and each entry is canonicalized to a 7-field tuple event: a unique event ID, process ID, thread ID, source file location, type, tag, value. Most fields are self-explanatory. The type field distinguishes acting and signaling events. We discuss tag and value fields in the next section. Samples are:

```
event(94,31,1,'Shell.java:189',act,thread,37)
event(95,31,1,'Client.java:226',sig,obj,484790)
```

HBs on events are essential for our task graph construction. Research has long focused on various designs of vector clocks and their variants in recording such relations. The space cost of timestamps based on vector clocks can become prohibitive when there are a large number of concurrent threads and processes or when inter-thread/inter-process communication is intensive, as a timestamp (whose size is decided by the number of threads and processes) has to be assigned to each event, and piggy-backed on thread and process interactions. While the resilience provided by vector clock based approaches is necessary for online testing or replaying concurrent applications, our goal is less stringent, which is to discover event structure.

We observe that the pair of events involved in a HB usually share common event data (fields). They may record the IDs of requests or objects on which the events execute. For example, a socket send event has a tag `sock` and value ⟨*FromIP*:*FromPort*, *ToIP*:*ToPort*⟩. Such information can be used to discover the corresponding receive event. In addition, the event type field indicates if an event is signaling or acting. Thus, for a pair of signaling and acting events from different threads with the same data tag and value, we infer that the signaling event happens before the acting event. For example, for the following events

```
event(68,3,32,'Client.java:149',sig,obj,280630)
event(69,3,1,'Client.java:724',act,obj,280630)
```

we can infer $e_{68} \rightarrow_e e_{69}$.

B. Prolog Based Event Processing

Our log processing and task graph construction techniques are based using Prolog, for three reasons. First, Prolog excels in inference over relations. It also provides the capability of recursive inference that is very desirable in our application, for example, in finding all reachable tasks from a given task. Second, although the techniques could also be implemented in an imperative language, the declarative style of Prolog allows us to easily generate various new relations from the existing ones. Third, Prolog implementation is concise and well-formed.

In the remainder of the paper, we will present our algorithms using Prolog predicates. Each predicate consists of two parts separated by the symbol "`:-`": the left hand side is called the *goal*; the right hand side is a set of conditions. The goal and the conditions are essentially relations. The meaning of the predicate is that an entry is created in

the goal relation if the set of conditions are satisfied. The variables are instantiated during the predicate evaluation. There are basic relations that do not have right hand side conditions. They are called *facts*. As shown in Listing 1, the `event` fact is used to define basic events as parsed from log files. It has 7 arguments or fields. The goal `hb` defines an HB between two events. It can be satisfied in a number of ways, exemplified by the two predicates with `hb` being the left hand side. The first predicate describes that any two events from different threads with matching data (`Tag,Value`) pair define an HB. The second predicate describes HBs caused by socket communications between threads or processes[2]. Relation `par_cond` describes pair of threads that are different. Note that simply using thread IDs is inadequate as different threads in different processes may have the same ID.

```
//Events are of form: (parsed from log files)
event(#,Process,Tid,Source,act|sig,Tag,Value).
//Event happens-before: event(X)→e event(Y)
hb(X, Y):- event(X, M, T, _, sig, Tag, Data),
    event(Y, M, S, _, act, Tag, Data),
    not S = T, not Tag = sock.
//happens−before caused by socket communication
hb(X, Y):- event(X, M, T, _, sig, sock, Data),
    event(Y, N, S, _, act, sock, Data),
    par_cond(M, T, N, S).
hb(X, Y):- ...

//true when thread T in process M and S in N
par_cond(M, T, N, S) :- M = N, not T = S.
par_cond(M, _, N, _) :- not M = N.

//Event graph formed by event happens−before
event_graph(G):- findall(X-Y, hb(X, Y), L),
    vertices_edges_to_ugraph([], L, G).

//Tasks: represented by two consecutive acting
//event ids from the same thread
task(Start, End):- event(Start,N,T,_,act,_,_),
    event(End,N,T,_,act,_,_),
    Start < End,
    not(event(Z,N,T,_,act,_,_), Start<Z, Z<End).

//Task happens-before: task(T,_)→t task(S,_)
task_hb(T,S):- task(T,E),task_has_sig(T,E,X),
    hb(X, S), task(S, _).
//task(B, E) has an signaling event S
task_has_sig(B,E,S):- task_event(B,E,S),
    event(S,_,_,_,sig,_,_).
//task(B, E) has event X in it
task_event(B,E,X):- event(B,N,T,_,_,_,_),
    event(X,N,T,_,_,_,_),
    B =< X, X < E.

//Task graph formed by task happens−before
task_graph(G):- findall(X-Y, task_hb(X, Y), L),
    vertices_edges_to_ugraph([], L, G).
```

Listing 1. Prolog task graph inference algorithm.

The remaining predicates in Listing 1 show how we can build a system-wide task graph by processing existing relations. The goal `event_graph` defines a directed graph with events being the vertices and all event HBs being the edges. Both `findall` and `vertices_edges_to_ugraph` are Prolog library predicates. The predicate binds a variable `L` by retrieving all `hb` edges and then transforms `L` to a graph. There are also library predicates finding paths or transitive closures in a graph. The goal `task` defines a task as discussed in Section III: a task is represented by the IDs of two consecutive acting events from the same thread. The goal `task_hb` defines HBs between tasks, namely, if there is a signaling event in the first task $[T, \_)$ happening before an acting event in the second task $[S, \_)$, then $T \rightarrow_t S$. Finally, the goal `task_graph` operates similar to `event_graph`.

C. False Positives/Negatives in HBs

As mentioned earlier, in order to constraint the overhead, our analysis infers tasks and HBs purely from logs. Since event logs are lossy, meaning they do not contain enough information to faithfully reconstruct what had happened during execution, our analysis has to handle false positives and false negatives.

False Positives Caused by Common Synchronization Objects. It is very common in Java programming to use the same synchronization object in multiple places, for instance, notify/wait on the same object may have multiple occurrences. Simply matching object ids in thread logs gives rise to false positives. Consider the example in Fig. 4. Thread `T1` notifies object o_1 at two places, Ⓐ and Ⓑ. The corresponding acting events are Ⓒ and Ⓓ, respectively. However, if we simply match the event parameters, false HBs are undesirably introduced between Ⓐ and Ⓓ, and between Ⓑ and Ⓒ.

```
preceding(X, Y):-
    event(X, M, T, _, sig, Tag, Data, Ts1),
    event(Y, M, S, _, act, Tag, Data, Ts2),
    not S = T, not Tag = sock, Ts1 < Ts2.
sig_preceding(X, Y):-
    event(X, M, T, _, sig, Tag, Data, Ts1),
    event(Y, M, S, _, sig, Tag, Data, Ts2),
    not S = T, not Tag = sock, Ts1 < Ts2.
hb(X, Y):- preceding(X,Y), not sig_preceding(X,_)
```

Listing 2. Prolog task graph inference algorithm.

Note that if there exists a global wall clock or a vector clock is used, the false positives can be eliminated. However, we already mentioned that vector clocks are too expensive in general. Realizing a global wall clock demands extracting the current timestamp, which entails a system call. We observe that the order of the events in the original process log file (before they are demultiplexed into per thread logs) can serve as logical timestamp. Now an event consists of 8 fields with an extra timestamp field *ts*. The revised rule for the HB relation regarding inter-thread interaction is shown in Listing 2.

Relation `preceding()` describes all pairs of signaling and acting events that operate on the same object and the signal event precedes the acting event. Relation

[2]In Prolog, '_' represents a wildcard

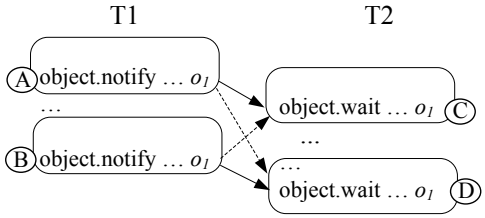


Figure 4. False positives (dotted edges) caused by synchronization objects.

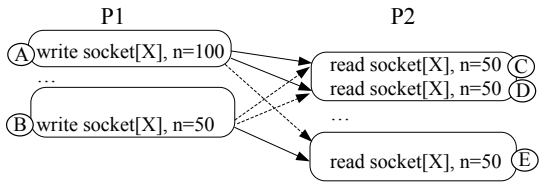


Figure 5. False positives caused by socket communication.

`sig_preceding` specifies all pairs of signaling events that operate on the same object and with the first event precedes the second event. Therefore, the HB relation is defined as a pair of signaling and acting events `X` and `Y`, with `X` preceding `Y`, excluding any signaling events that precede `X`.

False Positives Caused by Socket Communications. Another source of false positives is socket communication. The root cause is that while the OS level I/O is able to receive an entire packet sent by a socket write from a remote node, the JVM calls to socket reads often retrieve only a piece of the packet at a time. As a result, multiple sockets reads are the acting events corresponding to the single signaling event. However, allowing one to many mapping introduces false positives. Consider the example in Fig. 5. Node `P1` sends two packets using the socket. The packet sent at Ⓐ is read at Ⓒ and Ⓓ; the packet sent at Ⓑ is read at Ⓔ. Such scenarios create problems for us if we simply match event parameters: false positives are introduced from Ⓐ to Ⓔ and from Ⓑ to Ⓒ and Ⓓ. Note that we are dealing with logs from two different processes and hence the aforementioned timestamp idea is not applicable.

Our solution relies on the observation that the packet sizes in the multiple reads corresponding to a write aggregate to the size of the written packet. Hence, we maintain the number of bytes that have been read, if the size of the written packet has not been reached, the HB edges are still introduced from the same write. Otherwise, the algorithm moves on to the next written packet.

False Negatives Caused by Java's Guarded Wait Idiom. The intended semantics of Java `wait/notify` is as follows: a `notify` call unblocks a thread currently `waiting` on the object. However, a precisely conforming implementation is very hard, if not impossible, due to the existence of "spurious wakeups" (See the official Java API documentation on `java.lang.Object.wait(long)` for details). Spurious wakeups can unblock a thread when there has not been a call to `notify`. To deal with this condition, Java programmers are accustomed to write code like the following:

```
synchronized (ackQueue) {
  while(ackQueue.size() != 0) {
    try{
      ackQueue.wait();
      //log.info(''wait unblock''+ackQueue);/*A*/
    }catch(InterruptedException e) {}
  }
  //WaitMarker.markWait(ackQueue);          /*B*/
}
```

Namely, a `wait` is always coupled with a condition (here `ackQueue.size() != 0`). The meaning of the loop is that when a notification occurs, the size of `ackQueue` has to be 0. Otherwise, the unblocking of the `wait` is not caused by a notification, but rather a spurious wakeup. However, one side effect is that the `wait` may not even get executed, depending on the condition. In other words, the condition may play the role of synchronization in place of the `wait`. This poses a challenge to our event generation code inserted at statement *A*, which is supposed to emit an event after unblocking from `wait`. That is, an unblocking event is missed. The missed unblocking events will further cause missed event/task HBs when analyzing the logs (false negatives). To combat this problem, such wait idioms are singled out and a dummy API `WaitMarker.markWait(Object)` is introduced and placed right after the condition checking statement corresponding to each `wait`, as shown by statement *B* in the above example. We then change our AspectJ instrumentation code to log calls to `markWait` instead of `wait`. We also found that the rewriting burden is acceptable even for large systems like Hadoop, for which only 30 lines of code change are needed.

V. Evaluation

A. Performance

Our experiments are done on the Hadoop Distributed File System (HDFS), an open source project implemented in Java. It is designed to run on commodity hardware and supports MapReduce-style applications. The SVN checkout as of February, 2009 of the system contains 1558 Java source files, totaling 324K lines of code and 7.2M of class files (excluding libraries).

The original logging statements in the source code do not produce information meeting our requirements, i.e., they are insufficient to infer task graphs. Thus, we applied the logging aspects that we developed for Java-based distributed code, and run the resulting modified version of Hadoop. After weaving the aspects, the class file size grows to 8.8M, a 22.2% increase. The end-to-end request handling time increased by an average of 38% in instrumented version, compared with original version. This number is collected on an 8-node cluster on Emulab [10], with each node consisting of a 850MHz PIII CPU and 500MB memory.

Table II
TASK GRAPH BUILDING TIME IN PROLOG.

Properties	Log-A	Log-B	Log-C	Log-D	Log-E
# of events	2604	4768	6058	14042	21915
# of tasks	1359	2545	3339	5216	9872
# task hbs	597	2140	1867	6607	8488
time (s)	6	19	37	93	437

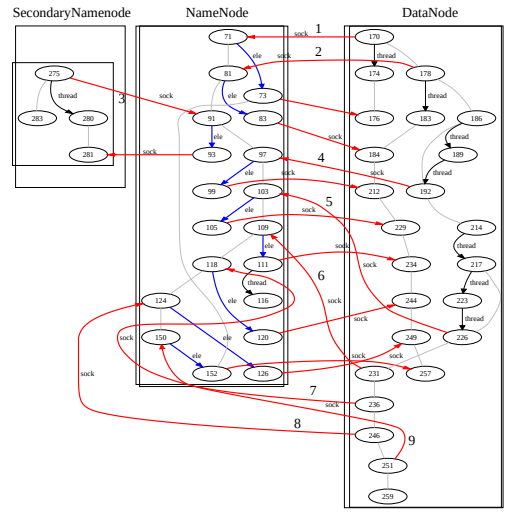
Table II shows the time that Prolog takes to infer the HBs and build the task graphs for some sample logs. These logs record between one to five minutes of Hadoop executions. The second to fourth rows represent, respectively, the number of events, the number of inferred tasks, and the number of happens-before relations in the logs. The final row represents the Prolog processing time in seconds; these numbers are collected on an AMD Opteron platform with two 2.4GHz CPUs and 6GB of RAM, running XSB Prolog Version 3.2.

B. Program Comprehension

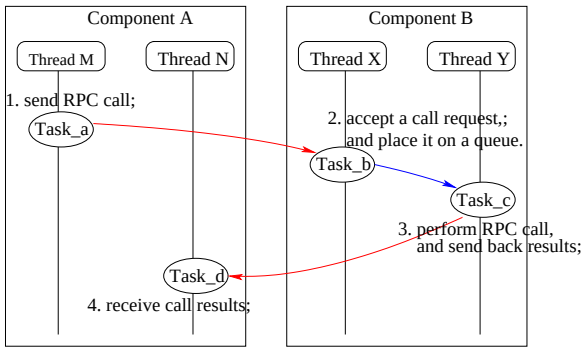
We demonstrate the utility of task graph in program comprehension as well as the flexibility of the query interface, using Hadoop HDFS. HDFS implements a set of file system APIs (PUT, GET, etc.) just like normal file systems. It transparently stores file data to hosts in a network. On a high level, its functionality is divided into three components: NameNode, DataNode, and SecondaryNamenode. More than one instance of these components can be configured to run on hosts in a network.

Panoramic view. When a new developer or maintainer joins the HDFS project, the first question to ask may be how NameNode, DataNode, SecondaryNamenode are started and interact with each other. To answer this question, we start HDFS with a simple configuration of one instance for each component, collect the logs, and build the task graph. Figure 6(a) shows a panoramic view of the obtained task graph depicting the interactions. This figure is produced by DOT from the popular GraphViz software after a simple conversion of the task graph data into DOT format. In this graph, nodes represent tasks; edges represent task HBs, with red edges denoting inter-process, blue edges denoting inter-thread, and black edges denoting thread spawn and reap; tasks are grouped by the component or process that executed them.

One can observe a re-occurrent task interaction pattern from Figure 6(a). The first interaction of each occurrence is labeled from 1 to 9 in the graph. This pattern is abstracted and shown in Figure 6(b). For example, the interaction $170 \rightarrow_t 71 \rightarrow_t 73 \rightarrow_t 176$ is an instance of the pattern. By examining the source code location attributes of the events in the tasks, we discover that they represent RPC calls, one of the main communication patterns in Hadoop. The 9 RPC calls in the graphs are shown in Table III. The RPC call names in the table suggest that at start-up, the very first thing DataNode and SecondaryNamenode do is to ensure



(a) Overall task interactions.



(b) Re-occurrent RPC call pattern from (a).

Figure 6. Task graph showing a panoramic view of the Hadoop HDFS component interactions. Nodes represent tasks; edges represent HBs.

they are talking through the same protocol with NameNode. Then, DataNode registers itself with the NameNode and reports the data blocks managed by it. The multiple calls of sendHeartbeat suggest that they are the keep-live messages between DataNode and NameNode. Indeed, the timestamp difference between consecutive heartbeat calls matches the value set in configuration file. These observations about the behavior of HDFS largely match those described in its development documentation.

#	From	To	RPC Call Name
1	DataNode	NameNode	getProtocolVersion
2	DataNode	NameNode	versionRequest
3	SecondaryNamenode	NameNode	getProtocolVersion
4	DataNode	NameNode	register
5	DataNode	NameNode	sendHeartbeat
6	DataNode	NameNode	blockReport
7	DataNode	NameNode	sendHeartbeat
8	DataNode	NameNode	sendHeartbeat
9	DataNode	NameNode	sendHeartbeat

Table III
HADOOP HDFS START-UP INTERACTIONS.

Closer look at distributed control flow. Another question that a developer or debugger may ask is how a distributed system proceeds after a certain "point". A point, for exam-

ple, can be an event indicating a fault or an event signifying the starting of a request. We show that task graphs are instrumental for answering such questions, using a case study of how replication in HDFS works. In HDFS, client files are broken into fixed-size data blocks; replication uses these data blocks as basic units.

This experiment is conducted on a HDFS cluster in Emulab with 7 hosts configured to run as DataNodes and one remaining host to run as the master NameNode (as well as SecondaryNamenode). We set the replication factor to three. We then wrote a file of twice the configured data block size to this cluster through the HDFS client, collected logs, and built the task graph.

To see how replication requests are handled, the following queries are performed:

```
findall([A, B], req_task(datanode, A, B), L).
reachable_tasks(A, B, Out) :- task_graph(G),
                  reachable([A,B], G, Out).
```

A `req_task(datanode,A,B)` is a task, `task(A,B)`, that starts with a socket receive event that does not have a matching socket send event (since the send event happened outside of the cluster in the client's code). Thus, the first query `findall(..)` will pick out all request starting tasks that executed in DataNodes and put these tasks in a list `L`. With our experiment set up, these tasks are the entry point of handling file replication requests. For all the tasks in `L`, we can run the second query to obtain all reachable tasks from `L`, and save the result in `Out`. The subgraph, *SubG*, formed by the tasks in `Out` are then converted to DOT file for visualization. *SubG* will show how the system proceeds after receiving the replication request.

Due to space limitation, the complete graph of *SubG* is not presented here. However, *SubG* consists of two disjoint subgraphs with similar interaction patterns, one of which is shown in Figure 7(a). Since the file written is of twice the data block size, one can infer that each subgraph of *SubG* may correspond to executions that replicate each of the two data blocks.

From the structure of Figure 7(a), we can gain some knowledge of how each data block is replicated: first, the data block replication request comes in to DataNode-1 at Task-3040 in the top left of Figure 7(a). Then, it gets forwarded to DataNode-2 and DataNode-3. The actual file block data is then received from the client and written at Task-3046 in DataNode-1 before being forwarded to the other two nodes. Finally, after receiving these data blocks, acknowledgments are propagated in the reverse direction, e.g. $1298 \rightarrow_t 3053$. To offer some confidence to this understanding of replication, Figure 7(b) shows the task graph with replication factor set to two.

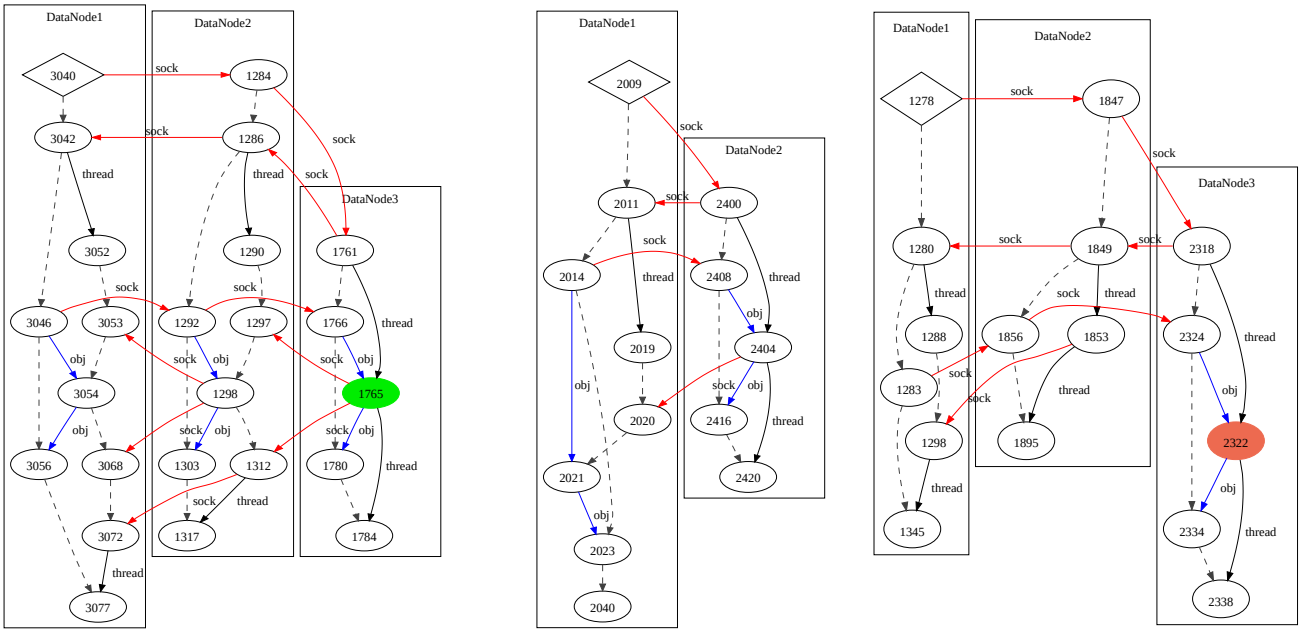
As a side effect, we also notice a potential inefficiency from the replication handling task graph: there seem to be two kinds of acknowledgments being propagated, as shown by the two different socket communications originated from Task-1765 at Node-3 (highlighted green; similarly from Task-1298 at Node-2). By querying the messages sizes as well as the source code locations of the corresponding events, we see that one acknowledgment is the packet sequence number of 8 bytes, and the other acknowledgment is a status value of two bytes. It seems that the sequence number acknowledgment alone is enough. Our communication on the Hadoop developer mailing list confirms this observation.

Lastly, we want to show that the task graph can also help understand the faults in the system. We rerun the experiment with replication set to three and force an fault by throwing an `IOException` at DataNode-3 in the replication chain. The exception is thrown in a `try-catch` block where such an `IOException` could have happened. The exception is thrown after DataNode-3 receives the data block, but before it acknowledges. Figure 7(c) shows the resulting task graph. Comparing it with Figure 7(a), we can clearly see that Task-2322 (highlighted red) in DataNode-3 in Figure 7(c) has behaved differently than the corresponding Task-1765 in DataNode-3 in Figure 7(a). To debug this fault, the developer can not only narrow down the fault location (by identifying the mal-aligned task), but also gain a understanding of its context, i.e., after receiving a data block and before acknowledging.

C. Request-Oriented Analyses

Modern distributed systems provide their services to clients through requests and replies. The goal of request-oriented analyses is to attribute runtime events and resource consumptions to each request or each request type. Data mining techniques can then be employed to find anomalies or inefficiencies. To be able to precisely attribute runtime data to individual requests, previous approaches assume that a unique ID is associated with each request and propagated throughout the entire execution serving the request. This assumption is unrealistic in practice as all the relevant data structures would need to be changed to accommodate the piggy-backed information. Changing the code is difficult in modern distributed applications as most of them use third-party library code. In this experiment, we illustrate that our inferred task graphs can be used to improve the precision of attributing runtime data to requests, which in turn reduce false positives (false alarms) in anomaly detection built on top.

Request-oriented analyses can be combined with a component interaction model to detect anomalies [6] (see Section VI). The premise of this technique is that a system's execution should behave similarly over time. One of the kinds of behaviors is how the components in the system interact with each other. Here, the number of interactions measures the number of communication operations over network sockets. Assuming that component A interacts with n other components, $B_i, i \in [1..n]$ and we measure the



(a) `replication=3` (b) `replication=2` (c) `replication=3, w/ Faults`

Figure 7. Replicating a data block under different scenarios. *The dotted edges serve to identify tasks in the same thread; they are not HBs.*

number of interactions between them over two time periods, t_1 and t_2, as $c_{1,i}$ and $c_{2,i}$, respectively, with $c_{x,i}$ representing the number of interactions between A and B_i over time period t_x, then the technique uses the following χ^2 value to quantify the behavior difference of time period t_2 compared to t_1:

$$\chi^2 = \Sigma_{j=1}^{N} \frac{(c_{2,j} - w_j)^2}{w_j}, \; with \; w_j = p_j C_2, \; p_j = \frac{c_{1,j}}{C_1}, \; and \; C_1 = \Sigma_{j=1}^{N} c_{1,j}, \; C_2 = \Sigma_{j=1}^{N} c_{2,j}$$

In particular, C_1 and C_2 represent the interactions of A with all other components; p_j is the ratio of the interactions between A and B_j over the total interactions in period t_1; w_j can be understood as the expected interactions in period t_2 if the same behavior pattern is assumed; χ^2 computes the standard deviation. Here, a higher value of χ^2 indicates a more significant divergence of the system behavior over time period t_2 from t_1.

Without task graphs, it is hard to know to which request an interaction should be attributed to, because answering a request might involve multiple processes and a process may be serving many requests in parallel. With task graphs, the origin request of an interaction can be correctly identified through graph reachability analysis (See Figure 3). We set up an experiment to demonstrate how task graphs improve the precision of the aforementioned χ^2 based anomaly detections. The experiment is conducted on a distributed storage system being developed at Microsoft. It is used to provide storage services for the Windows Azure cloud computing platform. It is built with features such as load balancing and fault tolerance. The application logic of the whole system is divided into many modules, called roles. Depending on the expected workload, these roles can be instantiated into a variable number of instances when deploying a live system. This system interacts with clients by supporting basic file system related operations: PUT operations that store client data into the system, GET operations that retrieve previously stored data, and a few others. Developers already have logging facilities implemented in the code, and hence we use the log files generated by the default setting. After collecting all the log files from all roles, events are identified and connected as described in Section IV.

In anomaly detection, it is a common practice to compare the system behavior of a later period to a previous period [6]. During these two periods, it is highly likely that the system will serve different mixtures of types of requests. To simulate this situation, we then collect logs for two runs: in *Run 1*, the system serves 750 PUT requests and 250 GET requests; in *Run 2*, the system serves 500 requests of each type. Since we use the same version of the system, we should expect any anomaly detection model to conclude that the system behavior during Run 2 is similar to Run 1, in other words, small χ^2 values should be expected.

Figure 8 summarizes our findings. We focus on a key component S in the system and observe its component interaction behavior with other roles. From the figure, we can see that if we apply the component interaction model naively (Figure 8(a)), the χ^2 value, 56.4, is quite large,

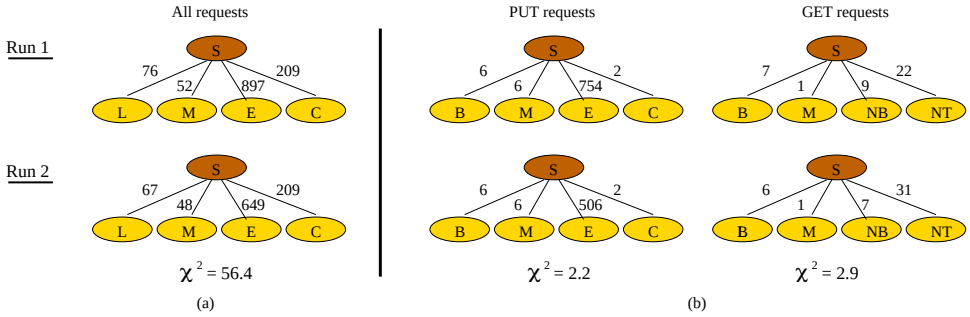


Figure 8. Improvements in the precision of component interaction model based anomaly detection. The numbers on the edges represent the number of interactions (a) applying the model naively; (b) applying the model after using task graph to classify the interactions for each request type.

suggesting anomaly. However, the large χ^2 is indeed caused by different request combinations being served. After using task graphs to attribute the interactions to different request types (Figure 8(b)), the χ^2 values are much smaller: 2.2 for serving PUT requests and 2.9 for GET requests, which suggests that the system behaviors during the two runs are similar, i.e., there are no anomalies. This latter result matches the experimental setup, suggesting that the attribution ability afforded by task graphs is critical to applying such models in practice and achieving results with low false positives.

VI. Related Work

Tracing. TraceBack [2] and Magpie [11], [12] also target the problem of understanding distributed applications. TraceBack aims to reconstruct basic block level or source line level control flows by using traces generated from static and dynamic instrumentation. Its focus is on merging traces from multiple threads into a single master trace. The difference with our work is that the distributed control flows that we aim to reconstruct are of coarser grain, i.e., tasks, which enables us to reason about systems consisting of large number of components and processes. Magpie generates request description strings by joining logged events serving the same request. These description strings record which components and resources are used in serving requests. The strings are then used to model workloads and detect anomalies. The work focuses on mining request strings and relies on domain knowledge to cluster events belonging to the same request. The work in this paper moves us closer to the goal of discovering request paths without relying on domain knowledge, by offering reusable tracing aspects for Java-based systems.

Other systems use a black-box approach to infer causal paths from protocol-level traces. Aguilera et al. [13] traces inter-node RPC messages, then, statistically infers causal paths offline and uses them to performance debugging. BorderPatrol [14] uses library call interceptors to generate traces, with every low overhead demonstrated in real systems (10-15%). To recover causal path, “a module designation identifies which request the module is currently processing”. Compared with our approach, both of these systems miss the causal relations brought by inter-thread signals.

Debugging. Several efforts aim at finding bugs in distributed systems by verifying *invariants* locally at some node or globally through data aggregated from multiple nodes. Research issues include choosing a logically consistent time to perform the checking, and developing scripts for developers to specify invariants [15], [16]. *General behavior* models leverage statistical analysis on large sets of system behavior data. Mirgorodskiy et al. studied the use of function-level traces in debugging with fail-stop and non-fail-stop failures in large systems with nodes running similar activities [17] (replicated systems). For each node, a time profile vector is built, summarizing percentages of time spent in each function (or call chain). Outlier nodes are found by using a distance measure for these vectors. Xu et al. [18] developed a general anomaly detection methodology for large-scale systems, in which they studied the effectiveness of applying the statistical method PCA, Principal Component Analysis, to feature matricies that were automatically constructed from console logs.

Zhang et al. [19] studied how to use basic metrics at system- and application-level to predicate *service level objective* (SLO) violations in three-tier architectures; in particular, how to adaptively select these metrics to be used in an ensemble of models for SLO violation detection. So instead of using just one model with a set of preset metrics to monitor the health of a distributed system, ensembles of models are used over time. They argue that as the system evolves, the system behavior might not be captured by the current model, so a new metric and new model need to be introduced to capture the evolved behavior. Their work is orthogonal to ours.

Cherkasova et al. [20] proposed a regression-based transaction model and an application performance signature model to detect application performance changes and distinguish these changes due to workload change from those due to performance anomaly.

Friday [21] is a replay based debugging system for distributed systems, capable of causally consistent group replay,

with each replayed node running inside a GDB process. A high-level script language is provided to break/watch/examine/update the distributed system as a whole; these commands are automatically translated into sets of normal GDB commands.

Visualization models. TotalView [22] is a parallel debugger. It can control multiple processes concurrently and offers rich UI for programmers to visually examine and change data arrays in MPI programs, for example. Arnold et al. [23] developed the Stack Trace Analysis Tool (STAT), which provides a visualization of a snapshot of the call stacks of all MPI processes by anchoring them in a tree structure. It is shown to be quite effective in finding tricky bugs in large networks running hundreds of MPI processes.

VII. Conclusion

This paper introduced *tasks* and *task graphs*. They can be used to analyze distributed systems the same way that basic blocks and CFGs are used to analyze sequential programs and their executions. They offer a box-and-arrow view of how distributed computation proceeds. Tasks cannot be mapped directly to programming language constructs for mainstream languages. Rather, they refine traditional operating systems concepts such as threads or processes and cut cross software engineering concepts such as classes or packages. We showed that task graphs are high-level enough to aid the understanding of the structure of distributed applications while the causal paths of the graphs help increase the accuracy of request-oriented anomaly detection.

Acknowledgment

This research is supported, in part, by the National Science Foundation (NSF) under grants 0847900 and 0834529. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] N. Leavitt, "Is Cloud Computing Really Ready for Prime Time?" *Computer*, vol. 42, no. 1, 2009.

[2] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel, "TraceBack: First Fault Diagnosis by Reconstruction of Distributed Control Flow," in *PLDI '05*.

[3] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[4] C. Fidge, "Timestamps in Message Passing Systems that Preserves Partial Ordering," in *11th Australian Computing Conference*, 1988, pp. 56–66.

[5] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: in search of the holy grail," *Distributed Computing*, vol. 7, no. 3, pp. 149–174, 1994.

[6] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, "Path-Based Failure and Evolution Management," in *NSDI '04*, pp. 309–322.

[7] V. S. Adve and R. Sakellariou, "Compiler synthesis of task graphs for parallel program performance prediction," in *LCPC 2000*, pp. 208–226.

[8] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an Object-Oriented Approach to Non-Uniform Cluster Computing," in *OOPSLA '05*, pp. 519–538.

[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *ECOOP '97*, pp. 220–242.

[10] "Emulab:network emulation testbed," http://www.emulab.net.

[11] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems," in *HotOS IX*, 2003.

[12] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for Request Extraction and Workload Modelling," in *OSDI'04*, pp. 259–272.

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

[14] E. Koskinen and J. Jannotti, "BorderPatrol: Isolating events for black-box tracing," in *EuroSys*, 2008, pp. 191–203.

[15] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code," in *NSDI '07*, pp. 243–256.

[16] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and X. Zhang, "D^3S: Debugging Deployed Distributed Systems," in *NSDI '08*.

[17] A. Mirgorodskiy, H. Maruyama, and B. Miller, "Problem Diagnosis in Large-scale Computing Environments," in *ICS 2006*.

[18] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *SOSP*. ACM, 2009, pp. 117–132.

[19] S. Zhang, I. Cohen, J. Symons, and A. Fox, "Ensembles of Models for Automated Diagnosis of System Performance Problems," in *DSN '05*, pp. 644–653.

[20] L. Cherkasova, K. Ozonat, M. Ningfang, J. Symons, and E. Smirni, "Anomaly? Application Change? Workload Change? Towards Automated Detection of Application Performance Anomaly and Change," in *DSN '08*.

[21] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global Comprehension for Distributed Replay," in *NSDI '07*, pp. 285–298.

[22] "TotalView debugger," http://www.totalviewtech.com.

[23] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *IPDPS*, 2007.