IBM Research Report

Mining Activity Data for Dynamic Dependency Discovery in e-Business Systems

Manoj K. Agarwal, Manish Gupta, Anindya Neogi
IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016, India.

Gautam Kar, Anca Sailer
IBM Research Division
IBM T.J. Watson Research Center
Hawthorne, New York, USA

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Mining Activity Data for Dynamic Dependency Discovery in e-Business Systems

Manoj K. Agarwal*, Manish Gupta*, Gautam Kar+, Anindya Neogi*, Anca Sailer+
*IBM India Research Lab, New Delhi +IBM T.J. Watson Research Center, New York

***Abstract*—The growing popularity of e-businesses has stimulated web sites to evolve from static content servers to complex tiered systems built from heterogeneous server platforms. A large amount of IT budget of e-businesses is spent nowadays on maintaining, troubleshooting, and optimizing these web sites. It has been shown that such system management activities may be simplified or automated to various extents if a dynamic dependency graph of the system were available. Currently, all known solutions to the dynamic dependency graph extraction problem are intrusive in nature, i.e. require unwelcome modifications at application or middleware level. In this paper, we develop non-intrusive techniques based on data mining, that process existing monitoring data generated by server platforms, to automatically extract the system component dependency graphs in tiered e-business platforms, without any additional application or system modification.**

***Index Terms*—Resource Management, Computer Network Management, Monitoring, Correlation**

I. INTRODUCTION

Web-based platforms have nowadays become extremely complex distributed systems with a large number of heterogeneous interacting components. From the point of view of an end-user interacting with a web-based storefront, seemingly simple "order" or "category browse" transactions are performed. However, tiers of back-end servers involving multiple server boxes and system components within them decompose and execute the sub-transactions in a coordinated fashion. Figure 1 illustrates a typical web-based stock brokerage implemented using various types of server platforms and system components.

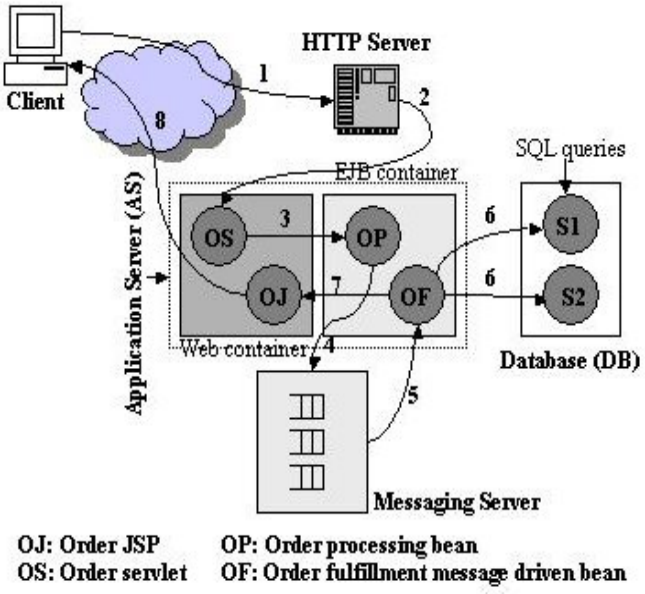

Fig. 1. Complex component dependencies in a stock brokerage application on J2EE/Messaging/DB platforms. The successive steps of an order transaction are indicated.

A typical complex path taken by an order transaction (e.g., "stock purchase") shows how servers and system components invoke other servers and components to execute the transaction. Since server vendors, application developers, system integrators, and system maintainers are typically from different organizations, there is no single entity with absolute knowledge of how the heterogeneous components in the deployed system run in synergy to execute user transactions. Hence, managing the deployed system, debugging problems, and improving end-to-end performance become exceedingly complex. For example, if the "stock purchase" transaction performance degrades drastically, the job of the system administrator turns into a tedious effort of manually

reading and correlating multiple logs to solve an end-to-end problem. Thus the cost of tuning or debugging of multi-vendor systems consumes a significant portion of enterprise IT budgets. Knowledge of end-to-end system dependencies enables semi-automated or even automated management applications, which use the information to perform cross-component analysis for pinpointing root causes of problems and end-to-end performance optimization [2, 7, 10, 15, 22].

The existing approaches for extracting and modeling system dependencies either fall short of enabling intelligent management applications or require significant system alteration, which prevents its wide spread use. An approach based on static analysis of application deployments, installation registries, or call graph analysis fails to capture the dynamic nature of dependencies in a complex system. In another approach, extra code is inserted into the application or middleware (called "instrumentation") to dynamically trace transaction paths and extract dependency knowledge. Even though the latter captures the dynamic nature of dependencies, organizations are often apprehensive about introducing extra code outside the development process, thus stalling the ubiquitous use of this approach. Additionally, middleware source code is often not available for end-to-end instrumentation in multi-vendor systems. Instrumentation framework standards [3] have been created, however, they have not been widely adopted for end-to-end transaction tracing.

This paper takes a data mining and statistical analysis approach to extract the dynamic component dependencies from *existing* system monitoring data. The central idea is to analyze the large amounts of usage and performance data already available from existing monitoring infrastructures on server platforms and extract dynamic dependencies between the monitored components, without introducing any *extra* instrumentation into the application or platform. The dynamic dependency information is stored and updated in a database accessible to dependency-based management applications. In this paper we propose two real-time dependency generation techniques and evaluate them in the context of well-known benchmark applications. Our approach is equally suitable for offline techniques, which are not presented in this paper.

II. Problem Statement

An e-business system can be modeled as a directed graph of interacting servers and system components at multiple levels of granularity. At the macro level, a node in the graph may represent a whole server, such as a database server or an application server. At a micro-level, a node may represent a component within a server, such as a servlet within the web container of an application server or a query on a database server. For example, in Figure 1, the dependency of the application server AS on the database server DB is a macro dependency, while the dependency of the EJB OF on the SQL query S1 is a micro dependency. In either case, a directed edge in our dependency model represents a service invocation or a dependent-antecedent relationship. In addition, weights are also attached to edges, representing dependency strengths. Dependency strength indicates the likelihood of the dependency being true. It is inherent to techniques based on statistical approaches, such as ours, to extract dependencies that do not actually exist (false dependencies). The objective of computing dependency strengths is to sort the dependencies of a node based on their weights, and cluster the true dependencies of the node towards one end of the list. A management application can use depth-first traversal order or select top N edges of each node to avoid false dependencies [20, 25].

We assume that the nodes of the system graph are known through mechanisms such as static analysis of code, deployment descriptors, and installation registries, platform-specific discovery techniques, or even manually ascertained domain knowledge. This is a valid assumption as we are monitoring the node performance and usage anyway. The aim of our project is to devise techniques so that existing monitoring data from these nodes may be used to infer the dynamic dependency relationships between them. It is assumed that each of the server platforms has a monitoring infrastructure in place. However, our techniques should not demand monitoring data beyond what is already available. Additionally, the level of detail in the system graph depends on the availability of detail monitoring data. For instance, if a component's monitoring data is not available then that component cannot be represented as a node in the system graph.

Among the various types of dependencies possible in an e-business system [16], we demonstrate our techniques specifically on user transaction to database query dependencies. The relevant monitoring data about user transactions, application server components, and database queries is used to find the dependency of individual user transactions on database queries. Transaction to query dependency captures micro dependencies between two system components: servlets handling transactions and queries handling back-end data access. It also captures macro dependencies because it exposes the dependency of the application server, hosting the servlet, on the database server, executing the query. Therefore, micro dependencies across servers automatically extract inter-box macro dependencies. The dependency extraction techniques discussed in this paper may be extended and/or can work with different techniques to extract other types of micro and macro level dependencies.

Different fault management and performance tuning applications may use the extracted dependency graph. The techniques to use the dependency graph in management applications are beyond the scope of this paper.

III. Related Work

A dependency graph of a system may be obtained using direct or indirect methods [4]. Direct methods rely on a human or a static analysis program to analyze system configuration, installation data, and application code to compute dependencies. However, such methods are unsuitable in large and heterogeneous systems because they are system specific and do not provide runtime dependency information. Indirect methods operate at runtime, and with respect to the manner they extract dependencies, they may be intrusive, semi-intrusive, or non-intrusive to the operational system. An example of an intrusive technique is one that relies on instrumentation such as ARM [3]. Dependencies are extracted by passing correlation IDs along with transaction flows, through instrumentation of application and/or middleware code. eWLM [2] is an IBM workload manager that relies on ARM instrumentation of the underlying components. Li [23] uses an instrumentation-based mechanism for global causality capture in HP printing systems. PinPoint [6] is a problem determination framework, where coarse-grained client requests are tagged as they travel through an enterprise system and discover the components. Tagging requires middleware-level instrumentation to pass the request ID between components, similar to ARM.

A key problem with the above approaches is that they may be unusable in situations where the system components are from multiple vendors or located in places where transaction correlation code cannot be inserted for security, licensing, or other technical constraints. Consequently, unless all components adhere to standards, such as ARM, instrumentation based approaches cannot be deployed as a full-fledged dependency generation solution. This motivates the requirement of semi or non-intrusive approaches. An example of a semi-intrusive approach is Active Dependency Discovery [5], where perturbation and fault-injection are used to infer dependencies.

In addition, Ensel [9] has also suggested the use of Neural Networks technology to automatically generate dynamic and cross-machine dependency graphs while monitoring is active. The technique however only detects correlation, without providing any evidence of causality. At the time of this writing, there are no details available regarding the training of such networks or to any experimental or theoretical accuracy and precision analysis of the method. Steinder et.al. [18] have also used the concept of belief networks for fault localization in network services built on complex communication topologies. The technique is specific to network services and the bipartite graph is developed specifically for problem determination.

Our approach falls in the non-intrusive category. We rely on the fact that most vendors provide some built-in instrumentation for monitoring statistics primarily for accounting or debugging purposes. These statistics provide at least process invocation or activity counters for monitored resources, and sometimes, even periods of actual usage of a resource by a transaction. We apply data mining techniques on the existing monitoring data to obtain probabilistic dependency information among resources used by transactions. Since we are not dependent on the actual "type" of the resource, the techniques may be easily generalized.

Hellerstein and Ma [11] have also applied data mining algorithms for discovering useful patterns in historical system event data. They examine how data mining can be used to identify actionable patterns; in particular they present algorithms for detecting three kinds of frequently occurring patterns in event data. Thoenen et al. [19], in the same context, have developed an event management tool along with a design methodology that have been widely used. The core of this methodology is a graphic representation of the roles and relationships between events.

A close match of our work is with Aguilera *et.al*. [20]. They use the message traces obtained from passive network sniffing to construct dynamic probabilistic dependency graphs between black boxes. The nodes that do not appear as source or destination in network messages or have addresses as part of encrypted payloads cannot be modeled. The work is complementary to ours because we rely on the nodes' monitoring data. We can also use nodes' activity traces obtained by passive network sniffing if the situation allows.

IV. Key Observations

The objective of this project is to extract runtime dependencies between monitored components in a web-enabled e-business system, more specifically, dependencies between transactions and database queries. We hypothesize that if a component is dependent on another component for completing a transaction, (e.g., "order"), then the periods when these components are actively used by different instances of "order" should be correlated. We define the activity period of a component as the period of time the component is performing some task when a transaction instance uses the component, like the execution period of an SQL query or a servlet. Thus, activity period is the difference between the start time when the component is invoked for a service and the end time when it finishes the service. We uncover correlations between these execution periods in order to derive the weighted dependencies between the components.

We work with two benchmark J2EE applications to investigate the efficacy of our techniques. The first one is a TPC-W compliant online bookstore application [21], which implements 14 types of user transactions related to "browse" or "order" categories. The other application, called Trade3 [24], captures the essence of an online stock brokerage application, where users can login to check stock quotes, buy, or sell stocks, and manage their investment portfolio. In this paper we investigate how effectively we can extract the user transaction to back-end database query dependencies in these applications using standard

system monitoring data. As we have mentioned previously, we require no application or middleware modification. Before describing our dependency generation techniques, we state some key observations that motivate these techniques.

We conduct the following experiment with TPC-W bookstore and a workload of 50 simultaneous customers running over 3 hrs. For each TPC-W transaction we record the start and end times of the transactions and all the database queries that are active when each of these transactions are active. We need not be concerned at this point about how these activity periods, i.e. start and end times, of transactions and queries are obtained as monitoring data from the system. We want to find out if there is underlying correlation between the activity periods of transactions and queries.

For each transaction type and query pair, we compute a ratio of the number of times the SQL query is concurrent with the transaction to the total occurrences of the transaction. We assume the strictest form of concurrency, i.e. the entire activity period of the query has to be *nested* within that of the transaction. In Figure 2, we show the plot of the ratio for the "bestsellers" transaction paired with all possible (41) database queries. It is interesting to observe that the ratio is much higher for antecedent-dependent pairs, such as {bestsellers, query #1} and {bestsellers, query #10}, compared to others. We know from studying the application source code that queries #1 and #10 are the only possible true dependencies for "bestsellers" transaction type. Thus true dependencies seem to have much higher ratio values compared to false or co-incidental dependencies that arise due to multithreading. This observation is exploited by our first technique to compute weighted dependencies between components.

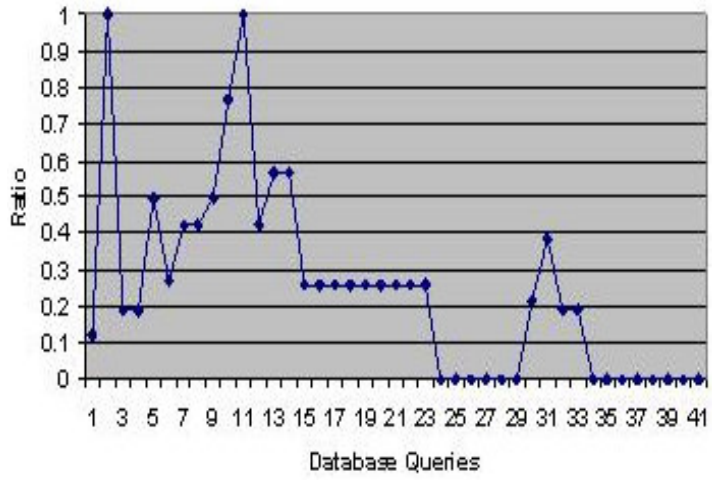

Fig. 2. Ratio of co-occurrence, i.e., the query is active when the transaction is active, for each database query paired with "bestsellers" transaction type. True dependencies show much higher ratios compared to false dependencies.

The nesting relationship in activity periods, of antecedents within dependents, can be applied to model correlation in synchronous transactions. However, component dependencies need not result from synchronous invocations only. For example, in Trade3, the order submission and order fulfillment business processes are decoupled and operate through a message queue, as shown in Figure 1. A user submits an order to buy or sell a stock and the transaction returns after submitting the order. The order fulfillment process picks up the order from a message queue and executes it. Thus, the activity period of a "buy" transaction has apparently no nesting relationship with the database queries that fulfill the purchase order. Asynchronous transactions require a different approach to derive the correlation between the activity periods of antecedent-dependent pairs. While in the case of synchronous transactions, the search is bounded by the dependent's response time; it is required to assume a worst-case window to search for an antecedent query of an asynchronous transaction.

We conduct a second experiment, with Trade3 this time, running a workload of 50 simultaneous customers for 1 hour to investigate how transaction to query occurrences are related, given a fixed search window after each transaction occurrence. For every transaction occurrence, we record its start time difference with the database queries that occurred within a worst-case search window of 1.5 seconds. We call this difference in start times, *transaction hop delay* or D_{TS}, where T is a transaction and S is a potential antecedent query that occurred in the search window of T. Figure 3(a) illustrates continuous D_{TS} samples made when an S lies within the 1.5 seconds search window of some T. The frequency distribution of D_{TS} for T="buy" and some database queries, S=S1, S3, and S5[1], is shown in the graph in Figure 3(b). Each y-value shows the percentage of total samples that have the corresponding start time offset on the x-axis.

Our prior knowledge about the application tells us that S1 and S2 are true dependencies and "buy" is not dependent on S5. Clearly, there is a difference in the D_{TS} *support* distributions between "buy" and these queries. Since occurrence of "buy" is not correlated with occurrence of S5, S5 occurs only a few times in the search window of a "buy" transaction when called by other concurrent transactions, but more importantly, the start time differences of S5 with "buy" are randomly separated when they coincidentally occur in the search window, leading to a uniform support distribution. On the other hand, S1 is consistently

[1] Sn is query number #n.

executed soon after "buy" starts, which explains the peak in the D_{TS} support distribution for the {"buy", S1} pair. The distribution for {"buy", S2} on the other hand shows multiple peaks (a multi-modal distribution) because conditional statements in the implementation of "buy" cause two different code paths to execute before invoking S2, leading to two major peaks.

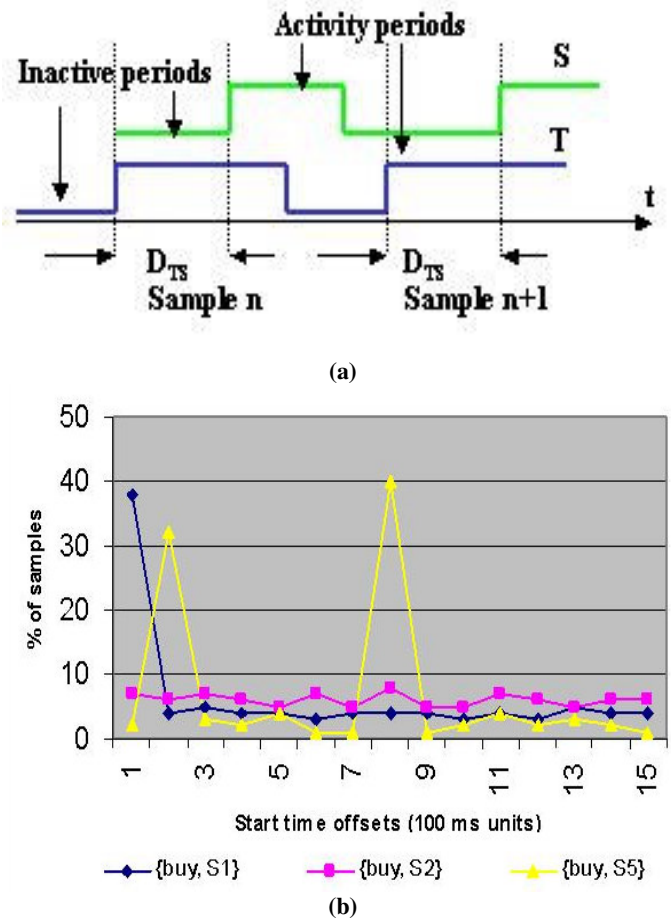

Fig. 3. (a) D_{TS} samples between transaction T and query S. (b) Distribution of start time offsets of transaction "buy" and three queries: S1, S2, and S5.

We can conclude from the experiment that true dependencies have skewed D_{TS} support distributions with one or more distinct peaks, while non-existent dependencies or coincidental occurrences of queries in search windows of transactions have relatively uniform distributions. We will exploit this difference of feature in D_{TS} support distributions between true and non-existent or false dependencies to attach dependency weights and order the dependencies in asynchronous transactions.

The final key observation is essentially a limitation in obtaining the monitoring data from some server platforms. Our techniques depend on the assumption that activity data of system components can be obtained from the heterogeneous monitoring infrastructure. However, most of the runtime monitoring infrastructures do not provide the detail activity traces of a component with respect to each transaction flowing through the system since it involves tremendous overhead. Thus, for some server platforms, activity data is in aggregate form. For example, the production-mode monitoring interface of the IBM WebSphere Application Server [12] can be queried to get only the number of instances of a servlet or EJB that were active in a given period. Therefore, sometimes, the dependency extraction techniques have to be resilient enough to work with aggregate data. Wherever only aggregate monitoring data is available, our platform-specific programs poll the local server-monitoring interface with some preset frequency to get progressive snapshots of the active components. Thus activity periods of components can be constructed within the observation error t, where t is the polling interval. Higher t produces coarser measurements, adversely affecting the precision of the statistical techniques, while lower t adversely affects system performance. Details about constructing approximate activity periods from aggregate data are described in Gupta *et.al.* [25].

V. TECHNIQUES

A. *Technique 1 for Synchronous Systems*

Our preliminary investigation, described in Section 4, indicated that the nesting rule of the activity periods might be used to compute dependencies in synchronous systems. Our first technique exploits this observation to assign edge weights to potential antecedent-dependent pairs. When outgoing dependency edges of a node are sorted based on this weight, the true dependencies are efficiently segregated from false dependencies.

The technique makes continuous observations of activity periods on the timeline. For example, Figure 4 shows a snapshot on the timeline with activity periods of components T1, T2, S1, S2, S3, S4 components, where T1 and T2 are transactions and S1, S2, S3, and S4 are database queries. The activity periods of transactions are measured by the application server transaction monitoring and the query activity periods are measured by database monitoring.

Let us assume that both the monitoring interfaces are polled at the same frequency, hence the start and end times of the activity periods have an observation error equal to the corresponding polling interval. If we know from static or domain knowledge that transactions T1 and T2 are both synchronous, then we can apply the nesting rule. From the snapshot in Figure 5, applying the rule, T1 is dependent on S1, S2 and S3. T2 is dependent on S3 and S4. The measurement and application of the nesting rule is a continuous process. Thus any transaction T is potentially dependent on any query S if at least one activity period of S is nested within any activity of T somewhere along the timeline. This leads to an un-weighted graph.

Since the system is multithreaded and transactions T1 and T2 may proceed simultaneously, it is quite possible that an antecedent query S3 of T2 executes when T1 is also executing, leading the nesting rule to wrongly infer a non-existent dependency between T1 and S3. We compute weights of each edge discovered by the nesting rule such that a higher weight is attached to a dependency, which is more likely to be true. A graph traversal routine, which visits a node's edges in descending order of their weight, is more likely to visit true dependencies before false dependencies.

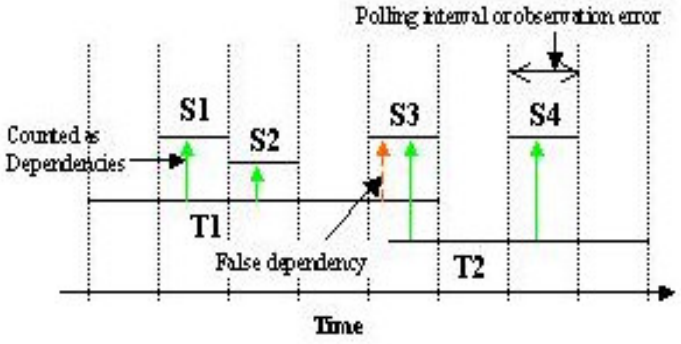

Fig. 4. Nesting rule for counting dependencies. Solid green lines are counted as dependencies and dotted red lines are false dependencies due to concurrency.

Our assumption, when attaching weights to discovered edges, is that false dependencies are a coincidence and occur less frequently in the long run. The weight of an edge T→S, also called the *forward probability*, is defined as *Prob(S|T)*, i.e. probability that the activity period of S is nested within the activity period of T. The probability is computed by the ratio *#{S,T}/#T*, where *#{S,T}* is the count of S occurring within T found by the nesting rule and *#T* is the total occurrences of T on the timeline. Forward probability is similar to the definition of "support" in the support-confidence framework of Agarwal *et.al.* [1].

Our experience with TPC-W and Trade3 has shown that the antecedent-dependent relationships are often not symmetric. For example, *#T* may be much higher than *#{S2,T}* because T calls several queries but S2 is called only by T. Thus, although T→S2 is a true dependency, *Prob(S2|T)* is small. In order to reduce the problem with weights computed from only forward probabilities, we also compute reverse probability of a T→S dependency as *Prob(T|S)*, i.e. probability that when S occurs, it is called from T. The reverse probability assigns a higher weight to asymmetric true dependencies such as T→S2.

We combine these two measures and come up with one single measure of the "likelihood" of each dependency edge. We conjecture that a dependency is more likely to be true if either of the measures is high, and an even stronger likelihood if both the measures are high. This statement is captured by the combined weight computation formula:

Edge weight = max (forward prob, reverse prob) + forward prob * reverse prob

Note that this weight computation technique is not foolproof in separating true and false dependencies when both forward and reverse probabilities are low for a true dependency.

B. Technique 2 for Asynchronous systems

Technique 1 may fail to capture all dependencies in case of asynchronous transactions, i.e. the callee's activity period is not necessarily nested within the caller's. Our second technique exploits the second observation we made in Section 4 about the difference in D_{TS} support distributions for true and false dependencies. In general, D_{TS} can be defined between any two consistently defined trigger times of activity periods of components T and S. Start times are a more natural choice.

In asynchronous transactions, the search window for antecedents of T is independent of T's activity period. In the absence of any good estimate of the temporal distance between a dependent and an antecedent, we pick an arbitrary worst-case search window, *w*. If any query S starts execution within *w* time units of the start of a transaction T, then T→S dependency is considered and weighed by technique 2. If this dependency were actually false, the weight assigned by technique 2 to T→S dependency edge would put it relatively lower down the sorted list.

We observed in Section 4 that false dependencies should ideally have a random D_{TS} support distribution and true dependencies should be skewed. The core idea behind technique 2 is to assign a weight to a T→S dependency, which quantifies how much its D_{TS} support distribution is "different" from a uniform distribution resulting from random D_{TS} samples. A higher weight signifies an increased likelihood of the dependency being true. We next explain how the D_{TS} distribution is built online and used to assign edge weights.

B.1 Building the D_{TS} distribution online

Assume there are *n* components *1, 2, 3,..., n*, which are either user transactions or database queries. The start time information for each component is obtained continuously from the monitoring interfaces of respective servers. The starting times are inaccurate by maximum observation error *t,* which is equal to the polling interval of the monitoring interface. Assume, for simplicity that *t* is the same for all monitoring interfaces.

Currently, we fix *w* as a multiple of *t, w=mt,* where *m* is provided as an input parameter. *m*, which may be fixed per transaction, signifies the maximum number of bins in the D_{TS} support distribution of a dependency pair. The following steps show how to build the D_{TS} support distribution for each T and S pair, where at least one of the activity periods of S starts within search window *w* after the start time of an activity period of T.

Step 1: The time line is chopped into intervals of length *t*, called *bins*. When a component's activity period starts, it is put into the nearest bin. A sliding window of only *m* bins, denoted as $b_0, \ldots, b_{m-1}$, is needed, corresponding to the search window *w*.
Step 2: For each bin b_j, we keep a count of the number of occurrences of component *c,* denoted as $r_c(j)$.
Step 3: For a pair of occurrences of components T (in bin *j*) and S (in bin *j+k*), separated by *k* bins, where *k=0, 1, 2, …, m-1*, the *support distribution* of {T, S}, is updated as follows:

$$D_{TS}(k) = \sum_{j=0}^{m-k} \min(r_T(j), r_S(j+k))$$

The D_{TS} distribution is updated with new monitoring data at each polling interval. The search window *w* is a sliding window over the trace of records.

B.2 Computing Edge Weights

If the hypothesis is that T does not depend on S, then the support distribution of D_{TS} should be uniform, because the random variable has equal probability to lie in any of the *m* bins. If the *measured distribution* is "*different*" from the uniform distribution, then the hypothesis is false and there exists a dependency between T and S. We use chi-square statistics to quantify the difference between the distributions in terms of chi-square probability. The computed chi-square value (χ^2), given by the equation below, is a measure of the *"difference"* between the hypothesized uniform distribution of *m* bins and the observed D_{TS} distribution.

$$\chi^2 = \sum_{k=0}^{m-1} (D_{TS}(k) - N/m)^2 \, m/N$$

where *N* is the sum of the *m* $D_{TS}(k)$ computed for the current search window. A higher χ^2 means more chance of dependency because the observed distribution is "*more different*" from the uniform distribution. The confidence to be placed on a chi-square value, *p* (the chi-square probability), is a well-known function of chi-square and sample-size. *p* is used as the *dependency strength*. Higher the sample-size, more confidence can be placed on a chi-square value. Dependencies of a resource are sorted in descending order of strength *p.* Thus, heavily skewed distributions with very few samples are assigned low weight.

The chi-square statistic depends on how the data is binned, i.e., the size of search window *w* and the polling interval *t*. At present, we fix *w* as a multiple of *t* in the absence of a more rigorous approach. When *w* is very large, it amounts to off-line analysis of a large amount of recorded monitoring data. Smaller *w* leads to near real-time analysis. *w* should be large enough to at least be able to capture the antecedents of each component in the search window and allow for enough number of bins to be

created ($m=w/t$). In our experiments we use $w=15t$. For higher precision, t should be small. However, from the monitoring system perspective, a higher t is desirable to reduce overheads. t is thus limited by the monitoring system.

VI. System Implementation

The techniques we described have been used to build a fully functional management system prototype for tiered IBM WebSphere/DB2 e-business system. We have used the extracted dependency graph to implement root cause analysis applications for response time related performance problems.

The system architecture is shown in Figure 5. The managed business system consists of an IBM HTTP server to handle URL requests from remote clients, an IBM WebSphere Application Server (WASv5.0) [12] for application offload, and a backend database IBM DB2 (v7.2) [14]. The WAS hosts a web container for servlets and JSPs, and an EJB container. TPC-W bookstore and Trade3 applications are installed on WAS with their data on the backend database. An embedded JMS messaging server is also installed on the WAS machine for asynchronous transactions. For example, the Trade3 order processing logic communicates with the order fulfillment process through a queue hosted on the messaging server.

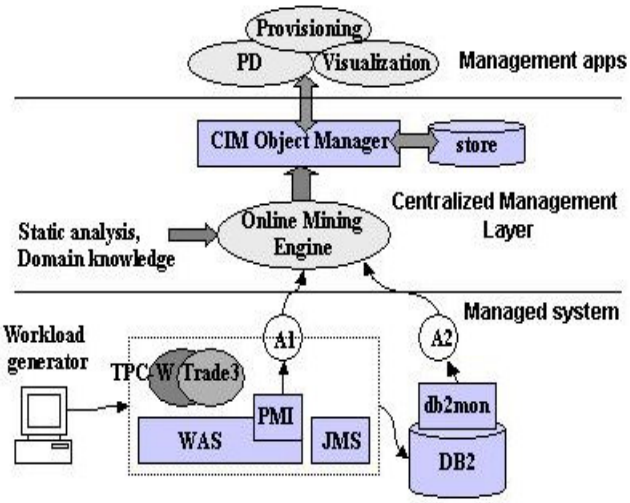

Fig. 5. Management system architecture: The management system has three tiers communicating using publish-subscribe messaging. WAS and DB2 are on 2GHz Pentiums with 512 MB RAM. For all the experiments WAS and DB2 have been put on the same machine. JMS server is always on the same machine as WAS.

The management system, also illustrated in Figure 5, has three tiers as well. The first tier consists of monitoring agents (see A1 and A2) specific to server platforms. These agents can interface with the monitoring APIs of the server platforms, extract component activity data and send them to the next tier. The second tier contains a centralized engine where the mining techniques are implemented. The engine computes the dependencies between transactions executed on the WAS and the SQL queries on the backend DB2 from WAS and database monitoring data. The correlation engine stores the extracted dependency information into a repository. A standardized object-oriented management data repository technology called Common Information Modeling (CIM) [8], is used to store the dependency information. CIM's standardized and open data access interface allows anyone to write a management application and use the continuously updated runtime dependency database, as shown in the top management application tier.

We utilize the Performance Monitoring Infrastructure (PMI) [17] provided by IBM WAS to extract transaction activity periods. Our monitoring agent program connects to the PMI interface and periodically polls the interface to get the total number of accesses and the average response time for each component on WAS. This information is used to reconstruct the activity periods of WAS components [25].

The DB2 monitoring data can be obtained by enabling tracing in the JDBC driver (in which case agent A2 is on the WAS machine), or by using the Snapshot interface provided by DB2 [14]. The JDBC tracing method is database independent and provides exact query activity periods w.r.t. the WAS box. The Snapshot API has to be polled similar to WAS PMI, it is database dependent, and can be used when JDBC tracing is either not possible or monitoring needs to be closer to the database.

If Snapshot API is used on DB2 and WAS is not on the same machine as DB2, then the activity information is aligned on a common timeline using our implementation of a lightweight clock delay estimation algorithm.

VII. PERFORMANCE EVALUATION

In this section we evaluate the performance of the dependency extraction techniques on our testbed.

A. Definition of Accuracy and Precision

TPC-W bookstore has 54 true transaction-to-query dependencies out of a potential set of 644 dependencies (14 transactions and 46 SQLs). Trade3 has 99 true dependencies out of a potential 264 (11 transactions and 24 SQLs). We evaluate the results of our experiments using two performance measures: *accuracy* and *precision*. *Accuracy* is defined as the percentage of the true dependencies discovered. *Precision* is defined for a node based on depth first graph traversal order typically used by problem determination applications [25]. In the sorted dependency list of a node having n outgoing edges, the edges are numbered (starting from the first edge) from 1 to n with m ($\leq n$) being the last true dependency in the list. We assign a penalty of m-i to the false dependency labeled i, where $1 \leq i \leq m$-1. The maximum possible penalty, when there are only false dependencies before the last true dependency, is therefore $w_{tot} := m(m-1)/2$. Thus the total penalty due to false dependencies is $w_f := \sum_{i<m,\, i \text{ is a false Dependency}} (m-i)$.

We use the percentage node precision defined as $100(1-\frac{w_f}{w_{tot}})$. Observe that this definition penalizes a false dependency more if it occurs higher in the list. The precision value reported in the following experiments is the *mean* percentage node-precision over all dependent nodes in the graph.

B. Obtaining Experiment Data

Before running experiments, we need to obtain the "ground truth" graph containing the exact set of dependencies between transactions and queries, so that we can compute accuracy and precision of the extracted graph.

For TPC-W bookstore, we instrumented the application source (similar to ARM [3]) in order to get the actual transaction to query dependencies. Trade3 application source is too complex for quick instrumentation, especially in EJB mode with the EJB container generating SQL queries. Therefore, we manually executed one transaction at a time through a browser and recorded the queries that were invoked (through JDBC driver tracing). In the absence of concurrency, exact dependencies were obtained. A transaction type was executed multiple times to make sure all code paths are executed. The union of the set of queries invoked represents the true dependencies of the respective transaction type.

TPC-W workload generation is performed using the Remote Browser Emulation package [21]. The number of simultaneous customers specifies the load in an experiment run. The think-time of requests is a uniform distribution with 7 seconds average for all experiments. The transaction transition matrix used is 50% "buy" and 50% "browse" from TPC-W spec, to exercise all parts of the application uniformly. For Trade3, we use the IBM Web Performance Toolkit (WPT) for load generation [26]. The `record` tool is used to record 400 user clicks manually executed through a browser. The `stress` tool is then used to replay the trace with varying number of simulated clients. The average think-time is 7 seconds with a uniform distribution and the browse to buy ratio is 1:1.

In both TPC-W and Trade3 the query traces are obtained from the JDBC driver. In TPC-W the transaction traces are obtained from PMI because a transaction type is handled by a unique servlet that can be monitored by PMI. In Trade3, all transactions are handled by the same servlet. Thus transaction traces are obtained from HTTP server monitoring data, where the `action` parameter of the logged URLs in `access.log` indicates the transaction type.

C. Accuracy and Precision

In all experiments, the extracted transaction to query graph stabilizes within 30 minutes, under constant workload. Thus, we run each experiment for 1 hour and compare the extracted graph to the ground truth to compute the accuracy and precision of the former. We take the average of several such experiments under same conditions to report a data point.

The most critical factor that impacts the performance of the techniques is the degree of concurrency and load on the system, which directly corresponds to the number of *simultaneous* customers using the web application. If multiple transactions proceed simultaneously, it is harder to separate true dependencies from false dependencies in bookstore using nesting relationship. Similarly, under high load in asynchronous environments, as in Trade3, the queuing delays within the system tend to make the D_{TS} support distributions more unstable. Thus true dependencies may be mistaken for false because they may have a more uniform support distribution (lowering accuracy) and false dependencies may exhibit some skew to be mistaken for true dependencies (lowering precision).

Figure 6 shows the variation of accuracy and precision values of TPC-W and Trade 3 with increasing customer load and concurrency. The number of customers is increased till a significant percentage of URL requests time out due to load. The WAS machine can support up to 200 customers with TPC-W and 125 customers with Trade3.

All Trade3 dependencies are obtained using technique 2, while TPC-W dependencies are obtained using technique 1. Accuracy is not a function of load. In fact a counting based approach, such as technique 1, is bound to give 100% accuracy because at least one occurrence of the dependency will be caught by the nesting rule independent of concurrency. However, technique 2 is not able to catch some dependencies, independent of load, because there are no consistent peaks in their D_{TS} support distributions at any load level. Accuracy improves from 5 to 25 customers in Trade3 because at very low loads the experiment needs to run longer to get enough samples for convergence.

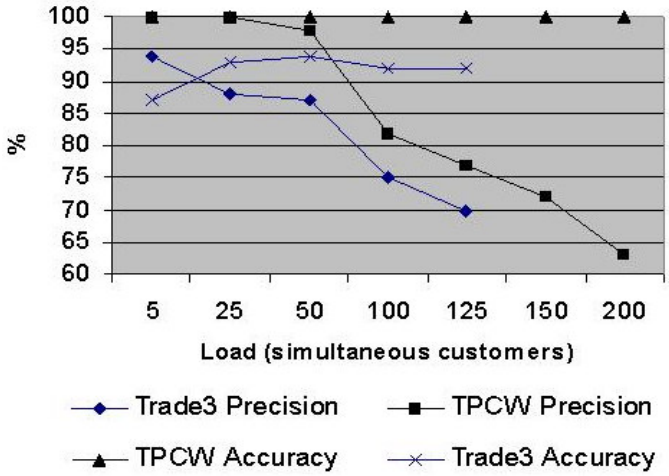

Fig. 6. Accuracy and precision with varying load for TPC-W/Technique1 and Trade3/Technique2 using 100ms polling interval of transaction activity.

As expected, both techniques show a drop in precision at higher loads. The precision of technique 2 is 6-12% lower than technique 1, which is expected because technique 2, in an attempt to generalize to asynchronous systems, removes the end-time constraint and assumes only start time information. However, at low loads both techniques give above 90% accuracy and precision, which means these techniques can be turned on when load levels are low and turned off when higher load is sensed.

We mentioned in Section 4 that many monitoring infrastructures provide only aggregate activity data through a polled interface. In our next experiment we investigate the effect of the polling interval or observation error t on the accuracy and precision of the techniques. Figure 7 shows the results.

In Figure 7 we only show Trade3/Technique2 measurements because TPC-W/Technique1 trends are similar [25]. The polling rate significantly affects precision and not accuracy (hence accuracy is not shown). At lower polling rates, the precision falls in both applications and using either technique. A higher polling rate is required to maintain high precision at the cost of higher overheads, as discussed next. The polling rate has less significance at lower loads because the transaction rates are lower.

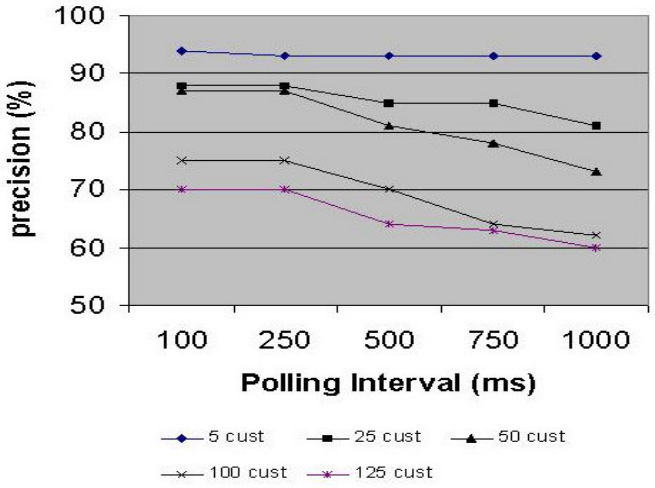

Fig. 7. Effect of PMI polling rate on precision in Trade3 using technique 2. Each plot is for a particular load level.

D. Scalability and Overheads

In both the techniques, the number of operations required to update the dependency graph for each dependent transaction instance T present in an activity trace is $O(n)$, where n is the number of instances of antecedents within the search window of T. At the same load level but higher polling rate, n may increase. Higher load levels will also increase n because transaction density increases. However, in all our experiments, we are able to update the dependency graph in real-time.

Ideally, dependency extraction should have no effect on the transaction data path beyond what is already introduced by an existing monitoring infrastructure. However, the agents that collect monitoring data may have some additional overhead. We perform end-to-end overhead measurements in terms of additional increase in average transaction response time when dependency extraction is active with a polling agent on WAS vs. when no dependency extraction is performed and the WAS agent is not active. In either case the basic WAS monitoring infrastructure and JDBC tracing are enabled. We vary the customer load and polling rate among data points. We show the results for TPC-W/Technique1 only because the overheads are independent of the application and extraction technique.

As shown in Figure 8, additional overhead of dependency extraction over and above normal monitoring overheads is mainly a function of polling interval and does not appreciably change with load. Higher polling rates increase the overhead because the agents run more frequently to query the monitoring interface, e.g. for agent A1 on WAS PMI, causing higher CPU consumption. This affects the transaction data path performance.

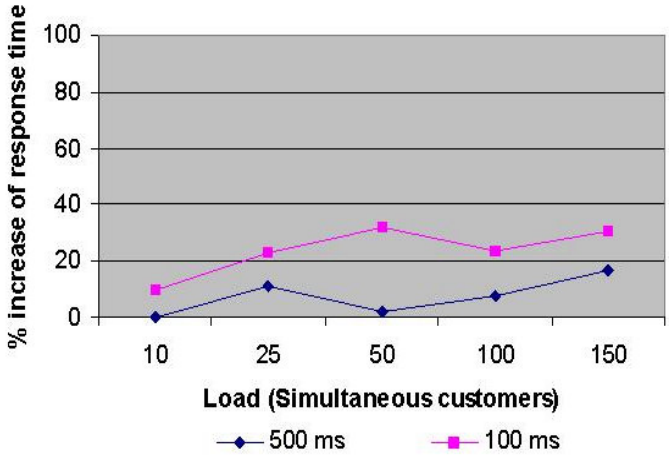

Fig. 8. End-to-end overhead measurements with and without dependency extraction at different polling intervals of WAS PMI. Basic monitoring infrastructure is anyway enabled all the time.

VIII. Summary and Future Work

In this paper, we applied data mining to extract resource dependencies from existing system monitoring data. The extracted dependency graph is an essential element for performance and fault management tasks. We described two techniques, for synchronous and asynchronous systems respectively, which perform with above 90% accuracy and precision under low loads on our testbed. We also measured the "non-intrusiveness" of our approach in terms of additional overhead (over and above existing monitoring) on the transaction data path.

In the future, we plan to extend our transaction to query dependency graph and include more servers and system components, such as JMS queues and DB server internals. This will also test our techniques in a more general distributed heterogeneous setting. Currently, we are also exploring technologies, such as root cause analysis, that can use the extracted dependency graph for better systems management.

References

[1] R Agarwal, T.Imielinski, and A. Swami, "Mining association rules between sets of items in large databases", *In Proc. of the ACM SIGMOD Conference on Management of Data, pages 207-216*, May 1993.

[2] J. Aman, C.K. Eilert, D. Emmes, P. Yocom, D. Dillenberger, "Adaptive Algorithms for managing a distributed data processing workload", *IBM Systems Journal, vol.36, no.2*, 1997.

[3] "Systems Management: Application Response Measurement", OpenGroup Technical Standard C807, UK ISBN 1-85912-211-6 July 1998, http://www.opengroup.org/products/publications/catalog/c807.htm.

[4] S. Bagchi, G. Kar, J.L. Hellerstein, "Dependency Analysis in Distributed Systems using Fault Injection: Application to Problem Determination in an e-commerce Environment" *12th International Workshop on Distributed Systems: Operations & Management* 2001.

[5] A. Brown, G. Kar, and A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in Distributed Environment", *IM* 2001.

[6] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services", *International Conference on Dependable Systems and Networks (DSN'02), June 2002*.

[7] J. Choi, M. Choi, and S. Lee, "An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes," *In 1999 IEEE International Conference on Communication*s, Vancouver, BC, Canada, 1999, pp. 1547–51.

[8] CIM: http://www.dmtf.org/standards/standard_cim.php.

[9] Ensel, Christian, "New Approach for Automated Generation of Service Dependency Models" *Second Latin American Network Operation and Management Symposium, LANOMS,* 2001.

[10] B. Grushke, "Integrated Event Management: Event Correlation using Dependency Graphs", *Proceedings of 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 98),* October 1998.

[11] J.L. Hellerstein and S. Ma, "Mining Event Data for Actionable Patterns", *The Computer Measurement Group*, 2000

[12] IBM WebSphere Application Server, http://www-3.ibm.com/software/webservers/appserv/

[13] A.K. Iyengar, M.S. Squillante, L. Zhang, "Analysis and Characterization of Large-Scale Web Server Access Patterns and Performance", *World Wide Web vol. 2 #1, 2*, June 1999.

[14] DB2: http://www-3.ibm.com/software/data/db2/

[15] S. Katker and M. Paterok, "Fault Isolation and Event Correlation for Integrated Fault Management", *Integrated Network Management V, Chapman and Hall*, May 1997.

[16] A. Keller and G. Kar, "Classification and Computation of Dependencies for Distributed Management", *5th IEEE Symposium on Computers and Communications (ISCC)*, July 2000.

[17] S. Rangaswamy, R. Willenborg, and W. Qiao, "Writing a Performance Monitoring Tool Using WebSphere Application Server's Performance Monitoring Infrastructure API", *In IBM WebSphere Developer Technical Journal,* February 2002.

[18] M. Steinder and A.S. Sethi, "Multi-layer Fault Localization using Probabilistic Inference in Bipartite Dependency Graphs", Technical Report 2001-02, CIS Dept., Univ. of Delaware, Feb 2001.

[19] D. Thoenen, J. Riosa, and J. L. Hellerstein, "Event Relationship Networks: A Framework for Action Oriented Analysis for Event Management," *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001, IEEE, New York (2001), pp. 593-606.

[20] M.K. Aguilera et.al., "Performance Debugging for Distributed Systems of Black Boxes", *19th ACM Symposium on Operating Systems Principles*, October 2003.

[21] TPCW Wisconsin website, http://www.ece.wisc.edu/~pharm/tpcw.shtml.

[22] S. Yemini, S. Kliger et al., "High Speed and Robust Event Correlation," *IEEE Communications Magazine*, vol. 34, no. (5), pp. 82–90, May 1996.

[23] J. Li, "Monitoring and Characterization of Component-Based Systems with Global Causality Capture", *23rd International Conference on Distributed Computing Systems*, May 2003.

[24] Trade3: http://www3.ibm.com/software/webservers/appserv/benchmark3.html

M. Gupta, A. Neogi, M. Agarwal, G. Kar, "Discovering Dynamic Dependencies in Enterprise Environments for Problem Determination", *Proceedings of 14th IFIP/IEEE International Workshop on Distributed Systems:Operations and Management (DSOM 03),* October 2003.

[25] Web Performance Toolkit: http://www.alphaworks.ibm.com/tech/wptools

Filename:	RI04002.doc
Directory:	C:\To Be Moved\Old_TP\Cyber
Template:	C:\Documents and Settings\Administrator\Application Data\Microsoft\Templates\Normal.dot
Title:	
Subject:	IEEE Transactions on Magnetics
Author:	-
Keywords:	
Comments:	
Creation Date:	6/8/2004 3:19 PM
Change Number:	2
Last Saved On:	6/8/2004 3:19 PM
Last Saved By:	IBM_User
Total Editing Time:	2 Minutes
Last Printed On:	6/8/2004 3:19 PM
As of Last Complete Printing	
Number of Pages:	13
Number of Words:	7,447 (approx.)
Number of Characters:	42,451 (approx.)