Adaptive Monitoring with Dynamic Differential Tracing-based Diagnosis

Mohammad A. Munawar, Thomas Reidemeister, Michael Jiang, Allen George, and Paul A.S. Ward*

Shoshin Distributed Systems Group
University of Waterloo, Waterloo, Ontario N2L 3G1,
{mamunawa, treideme, m4jiang, aageorge, pasward}@shoshin.uwaterloo.ca

Abstract. Ensuring high availability, adequate performance, and proper operation of enterprise software systems requires continuous monitoring. Today, most systems operate with minimal monitoring, typically based on service-level objectives (SLOs). Detailed metric-based monitoring is often too costly to use in production, while tracing is prohibitively expensive. Configuring monitoring when problems occur is a manual process.
In this paper we propose an alternative: Minimal monitoring with SLOs is used to detect errors. When an error is detected, detailed monitoring is automatically enabled to validate errors using invariant-correlation models. If validated, Application-Response-Measurement (ARM) tracing is dynamically activated on the faulty subsystem and a healthy peer to perform differential trace-data analysis and diagnosis.
Based on fault-injection experiments, we show that our system is effective; it correctly detected and validated errors caused by 14 out of 15 injected faults. Differential analysis of the trace data collected for 210 seconds allowed us to top-rank the faulty component in 80% of the cases. In the remaining cases the faulty component was ranked within the top-7 out of 81 components. We also demonstrate that the overhead of our system is low; given a false positive rate of one per hour, the overhead is less than 2.5%.

1 Introduction

Enterprise software systems are large and complex and their operators expect high availability and adequate performance. Proper operation given these conditions requires continuous monitoring. However, this increases operation costs since monitoring data is expensive to collect [1] and analyze [2]. System operators are faced with a choice of low-cost minimal monitoring, costly detailed monitoring, and prohibitively expensive tracing. Although detailed monitoring and tracing incur significant overhead, the information they provide is often essential for fault diagnosis. Given the performance ramifications, most enterprise

* The authors gratefully acknowledge the support of IBM and the Natural Sciences and Engineering Research Council of Canada (NSERC).

software systems today operate with minimal monitoring, using a few key metrics tied to service-level objectives (SLOs). When SLOs are violated a human operator enables detailed monitoring in an attempt to diagnose the fault, and, if unsuccessful, enables tracing. The quality of the final diagnosis is heavily dependent on operator skill, but they can err [3], and may be overwhelmed by the quantity of data collected.

We propose an alternative to this operator-driven approach: adaptive monitoring with dynamic tracing-based diagnosis. Our alternative has three steps: (1) error detection by monitoring of a minimal set of metrics *via* SLOs, (2) error verification by monitoring an extended set of metrics using invariant regression models, and (3) diagnosis using a differential analysis of ARM trace data collected from different peers. Each step is triggered based on analysis of data obtained in the previous step. Our approach is intended to keep the monitoring cost low during normal operation, only adding detailed monitoring and tracing when needed.

1.1 Background

Enterprise software systems comprise a mix of in-house, vendor-supplied, and third-party components, typically layered on standardized component frameworks like .NET [4], CORBA [5], J2EE [6], *etc.* To help operators monitor these systems, most components expose a variety of monitoring data at various granularity. Common data sources include performance metrics, correlated traces, and log records. In this paper we focus on J2EE-based systems, which provide monitoring data *via* management APIs such as Java Management eXtensions (JMX) [7] and ARM [8, 9]. Even though we focus on J2EE, the approach discussed here can be applied to other frameworks easily.

The JMX interface allows us to sample system metrics (*e.g.*, component response-time, activity count, resource pool status, *etc*) periodically. These are aggregate numerical values that reflect the state, behaviour, and performance of components. ARM traces contain fine-grained instance-level details such as order of component invocations and timing, which differ from the aggregate nature of metrics data. The order of component invocations can be combined to create complete paths. However, computing such paths in large-scale systems has a cost, especially when operations are logged out of order.

2 Approach

To illustrate our approach, we consider a simple J2EE cluster scenario, complete with load balancer, replicated application servers, and a common database back-end. We assume that the front-end load balancer can regulate the work assigned to a particular machine, and that admission control is in place to avoid system saturation. In addition, we assume that manifestation of multiple faults simultaneously in independent systems, is rare. As such, two faults, or even two instances of the same fault, are unlikely to be manifest in different application

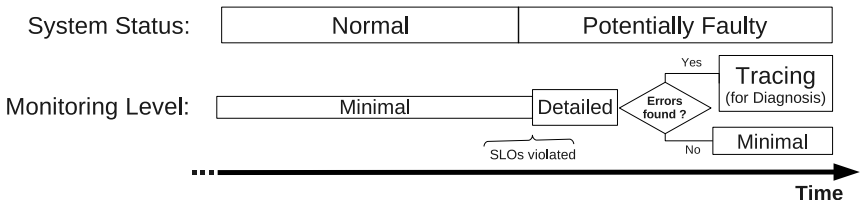

Fig. 1. Adaptive monitoring in action

servers at the same time. However, a fault in a shared subsystem (*e.g.*, database) could affect all dependent subsystems.

Any monitoring system has to balance complexity, overhead, and error-detection capability. This insight guides our adaptive scheme: error hypotheses are generated with minimal cost; these are consequently verified and diagnosed within only a short period of performance degradation. Our approach has three steps: *Error detection with minimal monitoring*, *error verification with detailed monitoring*, and *trace-based diagnosis*. The effective cost of our system is then a function of the false-positive rate. Figure 1 depicts the operating steps of our approach, while Algorithm 1 describes its logic.

Minimal monitoring entails tracking a small set of important metrics and detecting errors using SLOs. Because of its low overhead, we use it when system conditions are deemed normal. Error verification involves monitoring a larger set of metrics, and is thus more costly. We use invariant relationships between metrics to track this larger set. When both SLOs and invariants indicate an error, we enable tracing on the suspected subsystem and one of its peers deemed healthy. We then perform precise diagnosis by comparing the collected traces. As we discuss in Section 3.1, detailed monitoring is costly, it is used only when errors are detected in the first step. Tracing is costlier; it is enabled only when errors have both been detected with SLOs and validated with invariants.

2.1 Error Detection

Under normal conditions we only monitor a small set of carefully chosen metrics by comparing observed values against pre-specified SLOs in the form of thresholds. We check for SLO violations, which occur when the corresponding thresholds are consistently crossed. The monitored metrics are selected by system administrators based on four factors: they are sensitive to the state of a large subset of internal components, and so can detect a broad range of problems; they directly reflect users' perception of the service; they are inexpensive to measure and collect; and finally, problems that do not affect them are, by definition, not pressing enough to warrant further investigation. In our evaluation we monitor web page response times and number of failed requests of an Internet-based enterprise application. SLOs are typically defined based on contractual obligations (*i.e.*, service-level agreements) or user preferences. Alternatively, historical data

```
begin Monitoring
    mode := MINIMAL_MONITORING;
    while true do
        switch mode do
            case MINIMAL_MONITORING
                Monitor key metrics;
                if anomaly detected then
                    mode := DETAILED_MONITORING;

            case DETAILED_MONITORING
                Check invariant regression models;
                if error is confirmed for a cluster subsystem then
                    Reduce load submitted to the suspected faulty subsystem and a
                    non-faulty peer;
                    For both subsystems: mode := TRACING;
                if error is confirmed for all cluster members then
                    Report error in shared subsystems;
                if error is not confirmed then
                    mode := MINIMAL_MONITORING;

            case TRACING
                Compare trace data from the suspected faulty subsystem and a
                non-faulty peer;
                Diagnose based on the differences;

end
```

Algorithm 1: Pseudo-code of our approach

can be used to set SLOs such that a certain percentile of observations are within specified bounds.

2.2 Error Verification

While SLO monitoring can be inexpensive, its simplicity and reliance on static thresholds make it vulnerable to false alarms. False alarms not only require administrators' time, they also increase the monitoring overhead, as tracing is unnecessarily triggered. The error verification step aims to limit the monitoring cost that arises because of false alarms, while providing a robust means for validating the existence of an error.

Our verification step entails collecting a larger set of system metrics, among which stable, long-term correlations exist [10–13]. These correlations, also known as invariants, are captured *a priori* in the form of regression models using data collected from a healthy system. Each model associates two variables, one of which can be used to predict the other. As such, we can check each metric's behaviour by ensuring that its observed values are in line with predictions of the corresponding learned model. Verifying whether faults exist in the system involves determining the ratio of models that do not fit observations; when this ratio exceeds a specified level, the presence of faults is confirmed.

We have described our invariant-identification and error detection approach based on simple linear regression in previous work [12, 13]. Here we extend it to clustered systems, taking care to avoid identifying accidental correlations as invariants. Such correlations arise because of replication and coupling among subsystems. Replicated subsystems cause accidental correlations because they expose the same metrics and, because of the load balancer, experience similar

workloads; as such, metrics that correlate within the subsystem, also correlate between replicas. We therefore only learn correlations within replicas, rather than between replicas. Similarly, coupling is caused by shared subsystems such as the load balancer and the database. We avoid these correlations by discarding models that relate metrics of any subsystem to metrics of a shared subsystem.

Error verification allows us to pinpoint faulty subsystems and identify healthy subsystems. We do so by analyzing invariants on each subsystem and then comparing the results across peers. A healthy subsystem will exhibit none or few model violations. If a single peer is experiencing model violations, we presume it is the faulty peer; where multiple peers are experiencing violations, per our assumption of a single faulty subsystem, we presume there is a common subsystem causing the errors. The outcome allows downstream diagnosis engines to readily select a non-faulty cluster member as a baseline.

2.3 Diagnosis

When an error is reliably detected, we enable ARM tracing on two peers: one healthy and one suspected to be faulty. The detection and verification steps determine which peer is deemed healthy and which is not; a non-healthy peer would cause SLOs and invariant correlations to be violated. At this stage, equal amounts of requests are directed towards the two peers, ensuring their behaviour will be statistically similar, modulo the fault.

Timing Data: Many faults directly or indirectly affect the timing behaviour of individual components. In particular, such faults change the distribution of the time taken to complete operations. To diagnose timing-related faults, we compare sample distributions of operation time instances, obtained from traces, of a healthy subsystem to those of the suspected non-healthy one. We use the standard χ^2 Two Sample Test [14] to check whether the two distributions of time values are different for a given significance level, α. This test does not require us to assume any specific timing distribution or handle sample-size differences. If the timing samples of a component are found to be statistically different, it is further considered in the diagnosis.

Our diagnosis consists of a ranked list of components. The rank of a component-operation pair is based on a score given by Equation 1, which is the ratio of the average execution time in the two different peers:

$$S(C_k) = \frac{\max(\mu_1(C_k), \mu_2(C_k)) + 1}{\min(\mu_1(C_k), \mu_2(C_k)) + 1} \tag{1}$$

$\mu_1(C_k)$ and $\mu_2(C_k)$ are the means of the two sample distributions for each component-operation C_k, and 1 is added to ensure numeric stability. We take the ratio of the maximum over the minimum value so that the full range of changes between the two peers is captured.

Structural Data: We also use weighted component-connectivity graphs derived from traces, represented as coincidence matrices, to diagnose faulty components. Each edge represents a caller-callee relationship and the edge weight is determined by the number of times the relationship appears in traces. We use the following three properties of the coincidence matrices for diagnosis:

1. InCalls: Calls made to component C_k, $\mathrm{IC}(C_k) = \sum_{i=1}^{m} G(i,k)$.
2. OutCalls: Calls made by C_k to other components $\mathrm{OC}(C_k) = \sum_{i=1}^{n} G(k,i)$.
3. OutCalls–InCalls Ratio: $\mathrm{CC}(C_k) = \frac{\mathrm{OC}(C_k)}{\mathrm{IC}(C_k)}$.

We use the same anomaly scoring function as for timing data, *i.e.*, Equation 1 is applied to the structural measures $\mathrm{IC}(C_k)$, $\mathrm{OC}(C_k)$, and $\mathrm{CC}(C_k)$, to produce individual scores and rankings. Because we aggregate the structural data in the form of counts, we ignore all counts smaller than the minimal threshold t_{min}.

2.4 Diagnosis Integration

Each of the diagnosis methods focuses on distinct characteristics of system behaviour. We therefore combine their results to improve fault coverage. Let $\mathbb{C}$ be the set of all components, and $\mathbb{M}$ be the set of diagnosis methods. For each $c \in \mathbb{C}$, a method $m \in \mathbb{M}$ reports an anomaly score $s = m(c) \in [0, \infty)$. $m(c) = 0$ if m does not shortlist c as anomalous. The goal of integration is to combine individual scores, $m_i(c)$, into a global score, $\hat{s}$:

$$\hat{s}{:}(m_1(c), m_2(c), \ldots, m_k(c)) \rightarrow [0, \infty] \quad \forall c \in \mathbb{C} \quad m_1, m_2, \ldots, m_k \in \mathbb{M}$$

Simple Combination: We normalize the anomaly scores of the individual methods, m_i, to the range $[0, 1]$ and take their sum as the combined anomaly score $\hat{s}$.

$$\hat{s} = \sum_{i=1}^{k} m_i'(c) \; \forall c \in \mathbb{C} \qquad m_i'(c) : m_i(c) \rightarrow [0,1], i = 1, 2, ...k \; \forall c \in \mathbb{C}$$

Weighted combination: Some diagnosis methods can more accurately identify certain faults than others; this can be automatically determined *via* the diagnosis results. For example, faults that affect operation times can clearly be diagnosed by methods based on timing data; *i.e.*, there is a large gap in anomaly score between the faulty and non-faulty components. We thus give such methods more weight when their results indicate that they are accurate. Specifically, we weight the scores produced by the analysis of timing data by the ratio of the scores of the first- and second-ranked component. The other methods are assigned weights as described in the simple combination approach.

3 Evaluation

We set up a small clustered enterprise application environment to evaluate our adaptive-monitoring approach and differential trace-analysis methods. Our testbed, shown in Figure 2, comprises a DB2 UDB 8.2 database server, two WebSphere 6 Application Servers (WAS), workload generators, a monitoring engine,

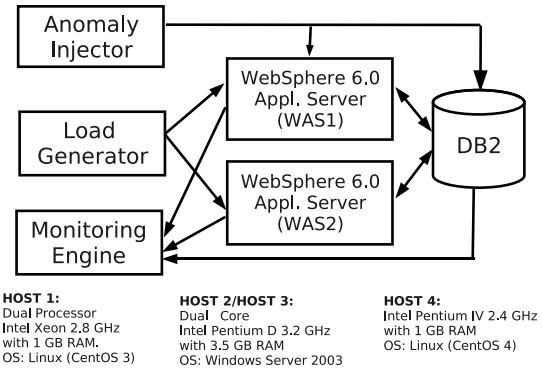

Fig. 2. Experimental Setup

and a fault-injection module. We use this infrastructure to execute the Trade benchmark [15], a J2EE application which implements an online stock-brokerage system.

The monitoring engine collects ARM and JMX data from the two application servers. The metric data is collected every 10 seconds from the JMX interface of the application servers and saved in a local database. For convenience, unless otherwise stated, the metric data is collected throughout experiments for offline analysis. Trace data is logged in files only when enabled; the files are then fetched by the monitoring engine for analysis. While we could transfer the trace data directly from ARM agents to the monitoring engine, we did not see a performance gain when compared to logging.

The load generator creates randomized workload patterns, subject to a maximum. Because we simulate user activity, we implemented the load-balancing logic as part of the load generation module. In practise, a separate workload balancer would distribute the work among cluster members. When tracing is enabled, the load-balancing logic reduces and controls the load such that a roughly equal amount of work is submitted to each application server.

3.1 Cost of Monitoring

We performed experiments to measure the overhead caused by the monitoring logic (*e.g.*, counter updates and time-stamping) and the data-collection logic. We used a simplified setup with one application server, a database, and an open-loop workload generator that enforces exponential inter-arrival time between requests. The service time in similar systems has been observed to follow the exponential distribution [16]. We thus model the system using an M/M/1 queue and derive the mean service time (T_s) thus:

$$T_s = \frac{1}{\mu} = \frac{T_r}{\lambda T_r + 1} \tag{2}$$

	Service Time		
Monitoring Level	*Mean (ms)*	*Std. Error (ms)*	*Overhead (%)*
None	7.468	0.001	0.0
Minimal	7.604	0.001	1.8
Detailed	10.152	0.009	35.9
Logged Trace	12.414	0.089	66.2

Table 1. Effect of monitoring on service time

where μ is the mean service rate, T_r is the response time, and λ is the request arrival rate. For each monitoring configuration, we execute experiments at different load levels. We repeat each experiment five times at every load level. For each experiment, our analysis takes 60 samples (10 minutes) into consideration; it excludes data collected during warm-up. Table 1 shows the mean service time as a function of increasing monitoring levels. The service times shown represent averages across the different load levels and the five repetitions. The standard error reflects the variation in the mean results. These results confirm that minimal monitoring has a small effect on performance, whereas detailed monitoring and tracing significantly degrade performance. While this increase may be acceptable for a brief period, it cannot be incurred continually. In particular, the overhead translates directly into the additional fraction of machines required for a data center with monitoring to service an equal load as one without, not taking into account analysis machines. Tracing has higher overhead than detailed monitoring. It also generates a larger amount of monitoring data which needs to be analyzed. Our analysis does not account for such overhead. These results support our claim that to contain the performance impact, detailed monitoring should only be used when an error is suspected, and that tracing should only be enabled if the error is confirmed.

Given these overhead numbers, we can estimate the cost of our adaptive monitoring scheme by assuming a false positive rate of less than one SLO triggered per hour, which is much more lenient than any system administrator would accept. With our setup and uniformly random workload, we have found that, using percentile-based SLOs determined from historical data, such a false alarm target can easily be achieved, while maintaining good fault coverage. Since detailed monitoring can refute the false positive in a minute, the mean service time of our approach would be $\frac{59}{60}7.604 + \frac{1}{60}10.152 = 7.646$, or an overhead of 2.39%.

Beside the measurement overhead, the analysis needed for our three-step approach incurs little computational overhead. During minimal monitoring, 20 metrics are tracked and each SLO can be checked in constant time. During detailed monitoring, the cost of analysis is $O(m \cdot i)$ where m is the number of invariant models and i is the number of sampling intervals considered. In our experiments m was close to 20000 for both application servers and detailed monitoring was enabled for 6 sampling intervals. The analysis needed for tracing depends on the method used. The cost of computing the InCalls and OutCalls measures is $O(c^2 \cdot s)$ where c is the number of components and s is number of

Parameter	*Value*	*Parameter*	*Value*
d_{len}	500 (ms)	d_{wait}	$[0, 100)$ (ms)
d_{max}	∞	e_{prob}	0.3
l_{interval}	1000 (ms)	l_{lock}	0.5

Table 2. Faults Parameters

entries per component in the traces. For analyzing the timing data, we need $O(c \cdot s)$ operations. In our experiments c, the number of components, was 81.

3.2 Faults and Fault-Injection Experiments

We have developed several types of faults to assess the effectiveness of our approach and methods. *Delay-loop faults* entail delaying completion of a selected method for d_{len} time units. To configure these faults, we specify a component, one of its methods, the delay-loop duration d_{len}, and a maximum number of loop instances d_{max}. These faults can be configured in two ways: (1) uniform-randomly spaced, by specifying a minimum random wait time between two delay-loop-executions d_{wait}; (2) probabilistic occurrences, by setting a probability of occurrence e_{prob}. *Exception faults*, with probability e_{prob}, throw an unhandled exception when a selected method is executed. A *table-locking fault* periodically locks a chosen database table. The lock is activated for l_{lock} fraction of every l_{interval} time interval during the fault-injection period. We have also developed a series of common operator-caused configuration faults, including authentication errors, incorrect thread-pool and connection-pool sizing, and component deletion. We do not evaluate these faults in this work because of space limits.

Each of our fault-injection experiments consists of a warm-up period, a period of normal activity during which we learn invariant regression models, and a period during which the system is monitored. We inject faults in the last period, while our monitoring is active. Unless stated otherwise, we use values listed in Table 2 to configure faults. The typical duration of an experiment is one hour.

3.3 Error Detection

During minimal monitoring, we oversee response-time and failure count metrics associated with all web pages of the Trade application. If either requests to a page fail or the response time of a page violates the corresponding SLO in three consecutive sampling intervals, we suspect presence of faults. Table 3 shows the effectiveness of our error detection approach. Each row represents an experiment where faults are injected in a method of a chosen component. We inject delay-loops and exception faults in components of WAS1. Table-locking faults are injected in the database.

The results show that all faults injected in WAS1 are detected. In the case of `QuoteBean`, we see SLO violations on the non-faulty application server (WAS2), which arise from the coupling induced by the database. We explain this further in Section 3.4. For table-locking faults, we see that, except for `ORDEREJB`, SLOs

Faults		Error Detection		Error Verification	
Type	Component	Detected on *WAS1*?	Detected on *WAS2*?	% Model with Outliers *WAS1*	% Model with Outliers *WAS2*
Exceptions	QuoteBean	Yes	No	4.8	0
	OrderBean	Yes	No	5.6	0
	HoldingBean	Yes	No	4.8	0
	AccountProfileBean	Yes	No	0.5	0
	AccountBean	Yes	No	1.7	0
Delay Loops	QuoteBean	Yes	Yes	2.1	0.01
	OrderBean	Yes	No	3.3	0
	HoldingBean	Yes	No	2.9	0
	AccountProfileBean	Yes	No	1.1	0
	AccountBean	Yes	No	2.7	0
Table Locking	QUOTEEJB	Yes	Yes	2.1	1.7
	HOLDINGEJB	Yes	Yes	0.6	1.3
	ORDEREJB	No	No	0.3	0.4
	ACCOUNTEJB	Yes	Yes	2	1.9
	ACCOUNTPROFILEEJB	Yes	Yes	1.8	1.5

Table 3. Error detection with SLOs and verification with invariant-correlation models

are violated on both WAS1 and WAS2. This is the expected behaviour, as both depend on the database.

3.4 Error Verification

In this work we only leverage intra-subsystem invariant correlations for each cluster member (*i.e.*, intra-WAS in our case). An example of such a correlation is shown in Figure 3 where the number of requests to the `TradeScenario` component is plotted against the number of store operation for the `OrderEJB` component for both WAS1 and WAS2.

The intra-WAS models allow us to perform diagnosis at the level of cluster members. A subsystem is faulty if a significant number of its invariants are violated. In practise, the administrator would decide what is significant based on data collected during validation of the invariants. Because the target system is dynamic with many sources of noise and the invariant models are statistical, we cannot expect all models to hold at all times. Thus, after a thorough validation, we expect none or a very small fraction of models to report outliers. Here, we consider faults to exist if 0.5% or more of intra-WAS models report outliers.

Table 3 summarizes results obtained from the analysis of correlation models for the experiments reported in Section 3.3. For both loop and exception-based faults, at least 0.5% of models within WAS1, the faulty application server, report outliers. Except for `QuoteBean` with loop-based faults, no model from WAS2 persistently reports outliers. Loop-based faults cause database connections to be tied up for longer periods and also cause the associated threads on the database to be unavailable. This limits the number of threads available to process work from WAS2. This problem is visible in the case of `QuoteBean`, as it is the most frequently used component of Trade. Nevertheless, very few models report outliers on WAS2 in this case. We are thus able to correctly confirm an error in WAS1 in all cases.

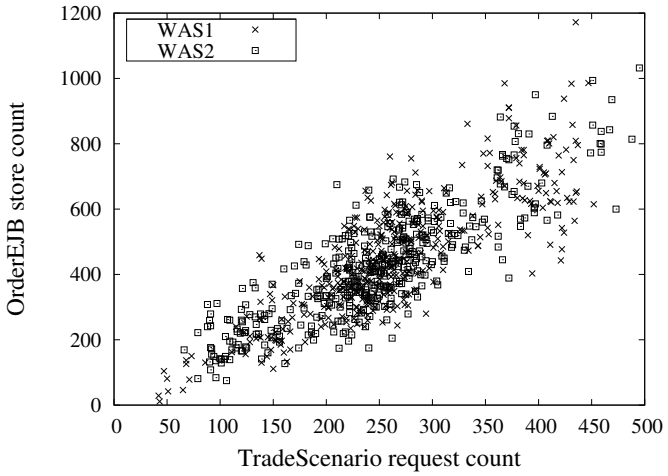

Fig. 3. Example of a stable correlation in both WAS1 and WAS2

Results for table-locking faults show a significant number of models reporting outliers on both WAS1 and WAS2 for all cases, indicating an error in the database. Tracing is thus not enabled, as it will not provide additional information on the status of the application servers. For the `ORDEREJB` table, the fraction of models with outliers is below the 0.5% threshold. In Table 3, we see that these faults are not detected using SLOs, thus neither detailed monitoring nor tracing is enabled. This case represents a false negative for our monitoring system.

3.5 Diagnosis

We now evaluate the accuracy of diagnosis methods and their combination. Table 4 summarizes the results obtained with the different methods. The first 10 results in Table 4 correspond to the first 10 results in Table 3; these represent experiments in which we apply all three steps, *i.e.*, detection, verification, and diagnosis if required. Note that diagnosis using the χ^2 test on the timing data only reports components whose operation time distribution changes at the 0.05 significance level.

We see two interesting phemomena in these results: (1) no one method is better than the rest, as the effectiveness of individual methods depends on the fault and the faulty component; (2) combining diagnoses can produce a better outcome than any single test; (3) the weighted combination generally beats the simple combination, but is sometimes (about 20% of the cases) worse than the best individual test.

For delay-loop faults, as expected, ranks using timing data are generally correct, while results with structural data are misleading. Given this, the weighted combination produces an exact diagnosis for the first set of experiments with delay-loop faults (lines 6–10), and only becomes weaker when the frequency of timing faults drops very low, to about 3% of the time (lines 16–20).

Fault Type	Faulty Component	Timing Data (OT)	OutCalls/ InCalls (CC)	InCalls (IC)	OutCalls (OC)	Simple Combination	Weighted Combination
Exceptions	QuoteBean.getDataBean	1	∞	1	∞	2	1
	OrderBean.getDataBean	9	2	1	3	1	1
	HoldingBean.getDataBean	7	2	6	2	1	1
	AccountProfileBean.getDataBean	1	∞	42	∞	9	6
	AccountBean.getDataBean	6	1	68	9	6	7
Delay loops	QuoteBean.getDataBean	1	∞	35	∞	7	1
$d_{\text{len}} = 500$ms	OrderBean.getDataBean	1	5	11	8	1	1
$d_{\text{wait}} = (0, 100)$ms	HoldingBean.getDataBean	1	∞	61	28	8	1
	AccountProfileBean.getDataBean	1	∞	39	∞	5	1
	AccountBean.getDataBean	1	4	20	13	1	1
Delay loops	QuoteBean.getDataBean	1	∞	48	∞	7	1
$d_{\text{len}} = 500$ms	OrderBean.getDataBean	1	4	32	13	2	1
$d_{\text{wait}} = (0, 1500)$ms	HoldingBean.getDataBean	1	∞	34	12	6	1
	AccountProfileBean.getDataBean	1	∞	16	∞	5	1
	AccountBean.getDataBean	1	3	13	8	1	1
Delay loops	QuoteBean.getDataBean	1	∞	20	∞	6	2
$d_{\text{len}} = 100$ms	OrderBean.getDataBean	1	4	33	20	3	1
$d_{\text{wait}} = (0, 7000)$ms	HoldingBean.getDataBean	3	∞	12	12	6	7
	AccountProfileBean.getDataBean	1	∞	11	∞	4	1
	AccountBean.getDataBean	1	4	32	16	2	1

Table 4. Ranks of faulty (component, operation) out of 81 possibilities for delay loop- and exception faults. A rank of 1 is the most accurate diagnosis, while a rank of ∞ means that the component is not short-listed

We perform additional experiments to evaluate the sensitivity of diagnosis based on trace data. The last 10 lines in Table 4 show the results of these experiments. We show that we are able to exactly diagnose delay-loop faults with loops lasting 500 ms with a random inter-loop gap with range $(0, 1500)$ ms, and within the top-7 even with a small disturbance of 100 ms with a random inter-loop gap with range of $(0, 7000)$ ms.

4 Related Work

Work in the area of error detection and diagnosis in enterprise software systems can be broadly divided into issues of data acquisition and analysis. Much work in data acquisition focuses on reducing monitoring overhead. Agarwala *et al.* [17] propose classes of channels, each with different rate and granularity of monitoring data; consumers can dynamically subscribe to these channels as needed. This approach only allows control of the communication overhead. Recent work on dynamic code instrumentation (*e.g.*, [18, 19]) focuses on how to efficiently adapt monitoring logic but not on when to do so.

Comparative analysis based on peer subsystems has been applied to offline diagnosis of known configuration faults (*e.g.*, [20, 21]). Pertet *et al.* [22] apply peer analysis to group communication protocols. But while their work presumes that the effects of the fault have spread, we assume efficient detection and validation, allowing timely and precise diagnosis. Kiciman and Fox [23] use peer comparison of paths to identify application-level faults. They, however, require continuous trace collection. Mirgorodskiy *et al.* [24] describe a methodology to compare timing behaviour of similar processes in a parallel computing environ-

ment. We focus on enterprise software systems which are often more dynamic than applications targeted in [24].

The use of invariant correlations between metrics for error detection and diagnosis was proposed both in our previous work [10] as well as by Jiang *et al.* [11]. The latter work assumes that a fixed set of metrics is always collected and no adaptation occurs. Agarwal *et al.* [25] also describe an approach to create fault signatures based on correlation between change-points in different metrics. Our prior work [12] is the first to demonstrate automated adaptive monitoring, and focuses on achieving the benefits of continuous monitoring at a fraction of the cost. The current work augments our earlier approach by diagnosing faulty components using more-precise trace data instead of metric-based invariants. Furthermore, the context of this work is a larger, clustered system, which allows us to employ novel techniques such as differential trace analysis.

Trace data analysis has been studied extensively. Kiciman and Fox [23] perform statistical comparison of instance-level component interactions. By contrast, our trace analysis is more efficient as it uses aggregate component interactions, and also looks at timing data. Kiciman and Fox also apply decision trees to correlate failures with faulty components. Likewise, Chen *et al.* [26] use clustering and Cohen *et al.* [27] use Bayesian models for the same purpose. Our work differs from these works in that they ignore monitoring costs.

5 Conclusions

In this paper we describe our approach to adaptively monitor software systems. Our approach consists of three, increasingly costly, steps: detection, verification, and diagnosis. To the best of our knowledge, it is the first monitoring system that automatically adapts from SLOs to tracing. It is also the first work that uses peer comparison of invariant models based on metrics for error detection. Unlike prior work that assumes continuous tracing, we enable detailed monitoring and tracing only when needed, thus incurring less than 2.5% overhead. Our verification approach uses peer comparison of invariant models of system metrics. We devise diagnosis methods based on differential analysis of information extracted from ARM traces and describe means to integrate their results. We show that once a statistically significant problem is detected, it is accurately validated and diagnosed. Our approach ensures that we only enable costly tracing when we are confident that tracing will accurately diagnose the defect.

References

1. Fox, A., Patterson, D.: Self-repairing computers. Scientific American (June 2003)
2. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer **36**(1) (2003) 41–50
3. Pertet, S., Narasimhan, P.: Causes of failure in web applications. Technical Report CMU-PDL-05-109, CMU Parallel Data Lab (December 2005)
4. Microsoft Corp: .NET Platform Available at http://www.microsoft.com/net/.

5. Object Management Group, Inc.: Common object request broker architecture (CORBA) http://www.corba.org/.
6. Sun Microsystems, Inc.: Java 2 platform enterprise edition, v 1.4 API specification http://java.sun.com/j2ee/1.4/docs/api/.
7. Sun Microsystems Inc.: JMX - Java Management Extensions. http://java.sun.com/javase/technologies/core/mntrmgmt/javamanagement/.
8. Johnson, M.W.: Monitoring and diagnosing application response time with ARM. In: SMW. (1998)
9. Rolia, J., Vetland, V.: Correlating resource demand information with arm data for application services. In: WOSP. (1998)
10. Munawar, M.A., Ward, P.A.: Adaptive monitoring in enterprise software systems. In: SysML. (June 2006)
11. Jiang, G., Chen, H., Yoshihira, K.: Discovering likely invariants of distributed transaction systems for autonomic system management. In: ICAC. (2006)
12. Munawar, M.A., Ward, P.A.S.: Leveraging many simple statistical models to adaptively monitor software systems. In: ISPA. (August 2007) 457–470
13. Munawar, M.A., Ward, P.A.: A comparative study of pairwise regression techniques for problem determination. In: CASCON. (2007) 152–166
14. Croarkin, C., Tobias, P., eds.: Engineering Statistics Handbook. National Institute of Standards and Technology (2006)
15. Coleman, J., Lau, T.: Set up and run a Trade6 benchmark with DB2 UDB. IBM developerWorks. http://www128.ibm.com/developerworks/edu/dm-dw-dm-0506lau.html?S_TACT=105AGX11&S_CMP=LIB.
16. Tesauro, G., Das, R., Jong, N.K.: Online performance management using hybrid reinforcement learning. In: Proceedings of SysML. (2006)
17. Agarwala, S., Chen, Y., Milojicic, D., Schwan, K.: QMON: QoS- and utility-aware monitoring in enterprise systems. In: ICAC. (2006)
18. Dmitriev, M.: Profiling java applications using code hotswapping and dynamic call graph revelation. In: WOSP. (2004) 139–150
19. Mirgorodskiy, A.V., Miller, B.P.: Autonomous analysis of interactive systems with self-propelled instrumentation. In: MMCN. (January 2005)
20. Mickens, J., Szummer, M., Narayanan, D.: Snitch: Interactive decision trees for troubleshooting misconfigurations. In: SysML. (April 2007)
21. Wang, H.J., Platt, J.C., Chen, Y., Zhang, R., Wang, Y.M.: Automatic misconfiguration troubleshooting with peerpressure. In: OSDI. (2004) 17–17
22. Pertet, S., Gandhi, R., Narasimhan, P.: Fingerpointing correlated failures in replicated systems. In: SysML. (April 2007)
23. Kiciman, E., Armando, F.: Detecting application-level failures in component-based internet services. IEEE Trans. on Neural Networks **16**(5) (Sept. 2005) 1027–1041
24. Mirgorodskiy, A.V., Maruyama, N., Miller, B.P.: Problem diagnosis in large-scale computing environments. In: Supercomputing Conference. (2006)
25. Agarwal, M., Anerousis, N., Gupta, M., Mann, V., Mummert, L., Sachindran, N.: Problem determination in enterprise middleware systems using change point correlation of time series data. In: NOMS. (April 2006)
26. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.A.: Pinpoint: Problem determination in large, dynamic internet services. In: DSN. (2002) 595–604
27. Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., Chase, J.: Correlating instrumentation data to system states: A building block for automated diagnosis and control. In: OSDI. (December 2004) 231–244